

*Un gato camina sobre el borde del tejado;
Tras él, el Sol se pone.
Gira su cara hacia mí, y creo que me sonrío.
Luego, de un salto, pasa sobre la línea del horizonte.*

Dedicado a Vivian, dondequiera que esté.

Ian ☸

INDICE

PRÓLOGO DEL AUTOR	17
CONTENIDO DEL LIBRO	17
AGRADECIMIENTOS	19
EL LENGUAJE SQL	21
<hr/>	
1. SISTEMAS DE BASES DE DATOS	23
ACERCA DEL ACCESO TRANSPARENTE A BASES DE DATOS	23
BASES DE DATOS RELACIONALES	24
INFORMACIÓN SEMÁNTICA = RESTRICCIONES	26
RESTRICCIONES DE UNICIDAD Y CLAVES PRIMARIAS	27
INTEGRIDAD REFERENCIAL	28
¿QUÉ TIENE DE MALO EL MODELO RELACIONAL?	29
BASES DE DATOS LOCALES Y SERVIDORES SQL	31
CARACTERÍSTICAS GENERALES DE LOS SISTEMAS SQL	33
EL FORMATO PARADOX	34
EL FORMATO DBF7	37
CRITERIOS PARA EVALUAR UN SERVIDOR SQL	39
INTERBASE	41
MICROSOFT SQL SERVER	43
ORACLE	45
OTROS SISTEMAS DE USO FRECUENTE	46
2. BREVE INTRODUCCIÓN A SQL	49
LA ESTRUCTURA DE SQL	49
PARA SEGUIR LOS EJEMPLOS DE ESTE LIBRO...	50
LA CREACIÓN Y CONEXIÓN A LA BASE DE DATOS	52
TIPOS DE DATOS EN SQL	53
REPRESENTACIÓN DE DATOS EN INTERBASE	54
CREACIÓN DE TABLAS	55
COLUMNAS CALCULADAS	56
VALORES POR OMISIÓN	57
RESTRICCIONES DE INTEGRIDAD	57
CLAVES PRIMARIAS Y ALTERNATIVAS	58
INTEGRIDAD REFERENCIAL	60
ACCIONES REFERENCIALES	61
NOMBRES PARA LAS RESTRICCIONES	62

4 La Cara Oculta de C++ Builder

DEFINICIÓN Y USO DE DOMINIOS	63
CREACIÓN DE ÍNDICES	64
MODIFICACIÓN DE TABLAS E ÍNDICES	65
CREACIÓN DE VISTAS	66
CREACIÓN DE USUARIOS	66
ASIGNACIÓN DE PRIVILEGIOS	68
ROLES	69
UN EJEMPLO COMPLETO DE <i>SCRIPT SQL</i>	70

3. CONSULTAS Y MODIFICACIONES 73

LA INSTRUCCIÓN SELECT: EL LENGUAJE DE CONSULTAS	73
LA CONDICIÓN DE SELECCIÓN	75
OPERADORES DE CADENAS	75
YO SÓLO QUIERO LOS DIEZ PRIMEROS...	76
EL VALOR NULO: ENFRENTÁNDONOS A LO DESCONOCIDO	77
ELIMINACIÓN DE DUPLICADOS	78
PRODUCTOS CARTESIANOS Y ENCuentROS	79
ORDENANDO LOS RESULTADOS	81
EL USO DE GRUPOS	82
FUNCIONES DE CONJUNTOS	83
LA CLÁUSULA HAVING	84
EL USO DE SINÓNIMOS PARA TABLAS	85
SUBCONSULTAS: SELECCIÓN ÚNICA	86
SUBCONSULTAS: LOS OPERADORES <i>IN</i> Y <i>EXISTS</i>	87
SUBCONSULTAS CORRELACIONADAS	88
EQUIVALENCIAS DE SUBCONSULTAS	89
ENCuentROS EXTERNOS	91
LA CURIOSA SINTAXIS DEL ENCuentRO INTERNO	93
LAS INSTRUCCIONES DE ACTUALIZACIÓN	94
LA SEMÁNTICA DE LA INSTRUCCIÓN UPDATE	95
VISTAS	96

4. PROCEDIMIENTOS ALMACENADOS Y TRIGGERS 99

¿PARA QUÉ USAR PROCEDIMIENTOS ALMACENADOS?	99
CÓMO SE UTILIZA UN PROCEDIMIENTO ALMACENADO	101
EL CARÁCTER DE TERMINACIÓN	102
PROCEDIMIENTOS ALMACENADOS EN INTERBASE	103
PROCEDIMIENTOS QUE DEVUELVEN UN CONJUNTO DE DATOS	106
RECORRIENDO UN CONJUNTO DE DATOS	108
TRIGGERS, O DISPARADORES	109
LAS VARIABLES <i>NEW</i> Y <i>OLD</i>	111
MÁS EJEMPLOS DE <i>TRIGGERS</i>	111
GENERADORES	113

SIMULANDO LA INTEGRIDAD REFERENCIAL	116
EXCEPCIONES	117
ALERTADORES DE EVENTOS	119
FUNCIONES DE USUARIO EN INTERBASE	121
5. TRANSACCIONES	125
¿POR QUÉ NECESITAMOS TRANSACCIONES?	125
EL ÁCIDO SABOR DE LAS TRANSACCIONES	127
TRANSACCIONES SQL Y EN BASES DE DATOS LOCALES	128
TRANSACCIONES IMPLÍCITAS Y EXPLÍCITAS	128
NIVELES DE AISLAMIENTO DE TRANSACCIONES	131
REGISTROS DE TRANSACCIONES Y BLOQUEOS	133
LECTURAS REPETIBLES MEDIANTE BLOQUEOS	136
VARIACIONES SOBRE EL TEMA DE BLOQUEOS	138
EL JARDÍN DE LOS SENDEROS QUE SE BIFURCAN	139
¿BLOQUEOS O VERSIONES?	141
NIVELES DE AISLAMIENTO Y TRANSACCIONES IMPLÍCITAS	143
6. MICROSOFT SQL SERVER	145
HERRAMIENTAS DE DESARROLLO EN EL CLIENTE	145
CREACIÓN DE BASES DE DATOS CON MS SQL SERVER	146
BASES DE DATOS EN LA VERSIÓN 7	148
TIPOS DE DATOS PREDEFINIDOS	149
TIPOS DE DATOS DEFINIDOS POR EL PROGRAMADOR	150
CREACIÓN DE TABLAS Y ATRIBUTOS DE COLUMNAS	151
INTEGRIDAD REFERENCIAL	152
ÍNDICES	153
SEGURIDAD EN MS SQL SERVER	153
PROCEDIMIENTOS ALMACENADOS	154
CURSORES	155
TRIGGERS EN TRANSACT-SQL	157
INTEGRIDAD REFERENCIAL MEDIANTE TRIGGERS	159
TRIGGERS ANIDADOS Y TRIGGERS RECURSIVOS	161
7. ORACLE	163
SOBREVIVIENDO A SQL*PLUS	163
INSTANCIAS, BASES DE DATOS, USUARIOS	165
TIPOS DE DATOS	166
CREACIÓN DE TABLAS	167
ÍNDICES EN ORACLE	168
ORGANIZACIÓN FÍSICA DE LAS TABLAS	169
PROCEDIMIENTOS ALMACENADOS EN PL/SQL	171
CONSULTAS RECURSIVAS	172

6 La Cara Oculta de C++ Builder

PLANES DE OPTIMIZACIÓN EN ORACLE	173
CURSORES	174
TRIGGERS EN PL/SQL	176
LA INVASIÓN DE LAS TABLAS MUTANTES	177
PAQUETES	179
ACTUALIZACIÓN DE VISTAS MEDIANTE TRIGGERS	181
SECUENCIAS	182
TIPOS DE OBJETOS	184

8. DB2 UNIVERSAL DATABASE 189

ARQUITECTURA Y PLATAFORMAS	189
AISLAMIENTO DE TRANSACCIONES	190
TIPOS DE DATOS	191
CREACIÓN DE TABLAS Y RESTRICCIONES	193
INDICES	194
TRIGGERS	195
CONSULTAS RECURSIVAS	196
PROCEDIMIENTOS ALMACENADOS	198

9. EL MOTOR DE DATOS DE BORLAND 199

QUÉ ES, Y CÓMO FUNCIONA	200
CONTROLADORES LOCALES Y SQL LINKS	201
ACCESO A FUENTES DE DATOS ODBC	202
¿DÓNDE SE INSTALA EL BDE?	202
EL ADMINISTRADOR DEL MOTOR DE DATOS	204
CONFIGURACIÓN DEL REGISTRO E INFORMACIÓN DE VERSIÓN	204
EL CONCEPTO DE ALIAS	206
PARÁMETROS DEL SISTEMA	206
PARÁMETROS DE LOS CONTROLADORES PARA BD LOCALES	208
BLOQUEOS OPORTUNISTAS	210
PARÁMETROS COMUNES A LOS CONTROLADORES SQL	211
CONFIGURACIÓN DE INTERBASE	214
CONFIGURACIÓN DE MS SQL SERVER	216
CONFIGURACIÓN DE ORACLE	217
CONFIGURACIÓN DE OTROS SISTEMAS	219
CREACIÓN DE ALIAS PARA BASES DE DATOS LOCALES Y SQL	219
ALTERNATIVAS AL MOTOR DE DATOS	220

C++ BUILDER: NAVEGACIÓN Y BÚSQUEDAS 221

10. CONJUNTOS DE DATOS: TABLAS 223

LA JERARQUÍA DE LOS CONJUNTOS DE DATOS	223
--	-----

LA ARQUITECTURA DE OBJETOS DEL MOTOR DE DATOS	225
¿TABLA O CONSULTA?	227
TABLAS (POR EL MOMENTO)	229
EXCLUSIVIDAD Y BLOQUEOS	231
CONEXIÓN CON COMPONENTES VISUALES	232
NAVEGANDO POR LAS FILAS	234
MARCAS DE POSICIÓN	235
ENCAPSULAMIENTO DE LA ITERACIÓN	236
LA RELACIÓN <i>MASTER/DETAIL</i>	239
NAVEGACIÓN Y RELACIONES <i>MASTER/DETAIL</i>	242
EL ESTADO DE UN CONJUNTO DE DATOS	247
11. ACCESO A CAMPOS	249
CREACIÓN DE COMPONENTES DE CAMPOS	249
CLASES DE CAMPOS	251
NOMBRE DEL CAMPO Y ETIQUETA DE VISUALIZACIÓN	253
ACCESO A LOS CAMPOS POR MEDIO DE LA TABLA	254
EXTRAYENDO INFORMACIÓN DE LOS CAMPOS	255
LAS MÁSCARAS DE FORMATO Y EDICIÓN	256
LOS EVENTOS DE FORMATO DE CAMPOS	258
CAMPOS CALCULADOS	259
CAMPOS DE BÚSQUEDA	261
LA CACHÉ DE BÚSQUEDA	263
EL ORDEN DE EVALUACIÓN DE LOS CAMPOS	264
EXTENSIONES PARA LOS TIPOS DE OBJETOS DE ORACLE 8	265
INFORMACIÓN SOBRE CAMPOS	268
CREACIÓN DE TABLAS	269
12. VALIDACIONES Y EL DICCIONARIO DE DATOS	273
VALIDACIÓN A NIVEL DE CAMPOS	273
PROPIEDADES DE VALIDACIÓN	274
EL DICCIONARIO DE DATOS	275
CONJUNTOS DE ATRIBUTOS	276
IMPORTANDO BASES DE DATOS	277
EVALUANDO RESTRICCIONES EN EL CLIENTE	278
13. CONTROLES DE DATOS Y FUENTES DE DATOS	281
CONTROLES <i>DATA-AWARE</i>	281
LOS ENLACES DE DATOS	283
CREACIÓN DE CONTROLES DE DATOS	284
LOS CUADROS DE EDICIÓN	285
EDITORES DE TEXTO	286
TEXTOS NO EDITABLES	287

COMBOS Y LISTAS CON CONTENIDO FIJO	287
COMBOS Y LISTAS DE BÚSQUEDA	290
ESENCIA Y APARIENCIA	292
CASILLAS DE VERIFICACIÓN Y GRUPOS DE BOTONES	292
IMÁGENES EXTRAÍDAS DE BASES DE DATOS	293
LA TÉCNICA DEL COMPONENTE DEL POBRE	293
PERMITIENDO LAS MODIFICACIONES	295
BLOB, BLOB, BLOB...	297
LA CLASE <i>TBLOBSTREAM</i>	298

14. REJILLAS Y BARRAS DE NAVEGACIÓN 301

EL USO Y ABUSO DE LAS REJILLAS	301
EL FUNCIONAMIENTO BÁSICO DE UNA REJILLA DE DATOS	302
OPCIONES DE REJILLAS	304
COLUMNAS A LA MEDIDA	304
GUARDAR Y RESTAURAR LOS ANCHOS DE COLUMNAS	307
LISTAS DESPLEGABLES Y BOTONES DE EDICIÓN	308
NÚMEROS VERDES Y NÚMEROS ROJOS	309
MÁS EVENTOS DE REJILLAS	312
LA BARRA DE DESPLAZAMIENTO DE LA REJILLA	313
REJILLAS DE SELECCIÓN MÚLTIPLE	313
BARRAS DE NAVEGACIÓN	314
HABÍA UNA VEZ UN USUARIO TORPE, MUY TORPE...	315
AYUDAS PARA NAVEGAR	316
EL COMPORTAMIENTO DE LA BARRA DE NAVEGACIÓN	316
REJILLAS DE CONTROLES	318

15. INDICES 321

CON QUÉ ÍNDICES PODEMOS CONTAR	321
ESPECIFICANDO EL ÍNDICE ACTIVO	323
ÍNDICES EN DBASE	325
ESPECIFICANDO UN ORDEN EN TABLAS SQL	326
BÚSQUEDA BASADA EN ÍNDICES	327
IMPLEMENTACIÓN DE REFERENCIAS MEDIANTE <i>FINDKEY</i>	329
BÚSQUEDAS UTILIZANDO <i>SETKEY</i>	330
EXPERIMENTANDO CON <i>SETKEY</i>	330
¿POR QUÉ EXISTE <i>SETKEY</i> ?	332
RANGOS: DESDE EL ALFA A LA OMEGA	333
EL EJEMPLO DE RANGOS DE CASI TODOS LOS LIBROS	335
MÁS PROBLEMAS CON LOS ÍNDICES DE DBASE	336
CÓMO CREAR UN ÍNDICE TEMPORAL	337

16. MÉTODOS DE BÚSQUEDA	341
FILTROS	341
ESTO NO LO DICE LA DOCUMENTACIÓN...	342
UN EJEMPLO CON FILTROS RÁPIDOS	343
EL EVENTO <i>ONFILTERRECORD</i>	346
LOCALIZACIÓN Y BÚSQUEDA	347
UN DIÁLOGO GENÉRICO DE LOCALIZACIÓN	350
FILTROS LATENTES	352
FILTER BY EXAMPLE	354
BÚSQUEDA EN UNA TABLA DE DETALLES	358
17. NAVEGACIÓN MEDIANTE CONSULTAS	361
EL COMPONENTE <i>TQUERY</i> COMO CONJUNTO DE DATOS	361
¿QUIÉN EJECUTA LAS INSTRUCCIONES?	362
CONSULTAS ACTUALIZABLES	363
SIEMPRE HACIA ADELANTE	365
CONSULTAS PARAMÉTRICAS	368
CONSULTAS DEPENDIENTES	370
LA PREPARACIÓN DE LA CONSULTA	371
VISUAL QUERY BUILDER	373
18. COMUNICACIÓN CLIENTE/SERVIDOR	377
NUESTRA ARMA LETAL: SQL MONITOR	377
APERTURA DE TABLAS Y CONSULTAS	378
LA CACHÉ DE ESQUEMAS	380
OPERACIONES DE NAVEGACIÓN SIMPLE	380
BÚSQUEDAS EXACTAS CON <i>LOCATE</i>	381
BÚSQUEDAS PARCIALES	382
UNA SOLUCIÓN PARA BÚSQUEDAS PARCIALES RÁPIDAS	383
BÚSQUEDAS CON FILTROS LATENTES	384
<u>C++ BUILDER: ACTUALIZACIONES Y CONCURRENCIA</u>	<u>387</u>
19. ACTUALIZACIONES	389
LOS ESTADOS DE EDICIÓN Y LOS MÉTODOS DE TRANSICIÓN	389
ASIGNACIONES A CAMPOS	390
CONFIRMANDO LAS ACTUALIZACIONES	392
DIFERENCIAS ENTRE <i>INSERT</i> Y <i>APPEND</i>	393
COMO POR AZAR...	394
MÉTODOS ABREVIADOS DE INSERCIÓN	395
EL GRAN EXPERIMENTO	396

10 La Cara Oculta de C++ Builder

EL GRAN EXPERIMENTO: TABLAS LOCALES	397
EL GRAN EXPERIMENTO: TABLAS SQL	398
PESIMISTAS Y OPTIMISTAS	399
EL MODO DE ACTUALIZACIÓN	400
LA RELECTURA DEL REGISTRO ACTUAL	402
ELIMINANDO REGISTROS	404
ACTUALIZACIÓN DIRECTA VS VARIABLES EN MEMORIA	404
AUTOMATIZANDO LA ENTRADA DE DATOS	406
ENTRADA DE DATOS CONTINUA	408

20. ACTUALIZACIONES MEDIANTE CONSULTAS 411

INSTRUCCIONES DEL DML	411
ALMACENAR EL RESULTADO DE UNA CONSULTA	412
¿EJECUTAR O ACTIVAR?	413
NUEVAMENTE COMO POR AZAR....	416
ACTUALIZACIÓN SOBRE CURSORES DE CONSULTAS	417
UTILIZANDO PROCEDIMIENTOS ALMACENADOS	418

21. EVENTOS DE TRANSICIÓN DE ESTADOS 421

CUANDO EL ESTADO CAMBIA...	421
REGLAS DE EMPRESA: ¿EN EL SERVIDOR O EN EL CLIENTE?	422
INICIALIZACIÓN DE REGISTROS: EL EVENTO <i>ONNEWRECORD</i>	423
VALIDACIONES A NIVEL DE REGISTROS	424
ANTES Y DESPUÉS DE UNA MODIFICACIÓN	425
PROPAGACIÓN DE CAMBIOS EN CASCADA	427
ACTUALIZACIONES COORDINADAS MASTER/DETAIL	428
ANTES Y DESPUÉS DE LA APERTURA DE UNA TABLA	429
TIRANDO DE LA CADENA	430
LOS EVENTOS DE DETECCIÓN DE ERRORES	431
LA ESTRUCTURA DE LA EXCEPCIÓN <i>EDBENGINEERROR</i>	432
APLICACIONES DE LOS EVENTOS DE ERRORES	436
UNA VEZ MÁS, LA ORIENTACIÓN A OBJETOS...	438

22. BASES DE DATOS Y TRANSACCIONES 439

EL COMPONENTE <i>TDATABASE</i>	439
OBJETOS DE BASES DE DATOS PERSISTENTES	440
CAMBIANDO UN ALIAS DINÁMICAMENTE	441
BASES DE DATOS Y CONJUNTOS DE DATOS	443
PARÁMETROS DE CONEXIÓN	444
LA PETICIÓN DE CONTRASEÑAS	445
EL DIRECTORIO TEMPORAL DE WINDOWS	447
COMPARTIENDO LA CONEXIÓN	448
CONTROL EXPLÍCITO DE TRANSACCIONES	449

ENTRADA DE DATOS Y TRANSACCIONES	450
23. SESIONES	453
¿PARA QUÉ SIRVEN LAS SESIONES?	453
ESPECIFICANDO LA SESIÓN	454
CADA SESIÓN ES UN USUARIO	454
EL INICIO DE SESIÓN Y LA INICIALIZACIÓN DEL BDE	455
SESIONES E HILOS PARALELOS	457
INFORMACIÓN SOBRE ESQUEMAS	460
EL MINIEXPLORADOR DE BASES DE DATOS	461
GESTIÓN DE ALIAS A TRAVÉS DE <i>TSESSION</i>	463
DIRECTORIOS PRIVADOS, DE RED Y CONTRASEÑAS	464
24. ACTUALIZACIONES EN CACHÉ	467
¿CACHÉ PARA QUÉ?	467
ACTIVACIÓN DE LAS ACTUALIZACIONES EN CACHÉ	468
CONFIRMACIÓN DE LAS ACTUALIZACIONES	469
MARCHA ATRÁS	471
EL ESTADO DE ACTUALIZACIÓN	472
EL FILTRO DE TIPOS DE REGISTROS	473
UN EJEMPLO INTEGRAL	474
EL GRAN FINAL: EDICIÓN Y ENTRADA DE DATOS	476
COMBINANDO LA CACHÉ CON GRABACIONES DIRECTAS	478
PROTOTIPOS Y MÉTODOS VIRTUALES	482
CÓMO ACTUALIZAR CONSULTAS “NO” ACTUALIZABLES	483
EL EVENTO <i>ONUPDATERECORD</i>	486
DETECCIÓN DE ERRORES DURANTE LA GRABACIÓN	487
¿TABLAS ... O CONSULTAS EN CACHÉ?	489
<u>PROGRAMACIÓN DISTRIBUIDA</u>	491
25. CONJUNTOS DE DATOS CLIENTES	493
CREACIÓN DE CONJUNTOS DE DATOS	493
CÓMO EL <i>TCLIENTDATASET</i> OBTIENE SUS DATOS	495
NAVEGACIÓN, BÚSQUEDA Y SELECCIÓN	496
FILTROS	497
EDICIÓN DE DATOS	498
CONJUNTOS DE DATOS ANIDADOS	499
CAMPOS CALCULADOS INTERNOS	502
ÍNDICES, GRUPOS Y VALORES AGREGADOS	503

26. EL MODELO DE OBJETOS COMPONENTES	507
UN MODELO BINARIO DE OBJETOS	507
¡YO QUIERO VER CÓDIGO!	508
CLASES, OBJETOS E INTERFACES	509
EL LENGUAJE DE DESCRIPCIÓN DE INTERFACES	511
IDENTIFICADORES GLOBALES ÚNICOS	513
INTERFACES	514
LA INTERFAZ <i>IUNKNOWN</i>	516
TIEMPO DE VIDA	517
INTROSPECCIÓN	518
CÓMO OBTENER UN OBJETO COM	520
PUNTEROS INTELIGENTES A INTERFACES	521
27. SERVIDORES COM	525
INTERCEPTANDO OPERACIONES EN DIRECTORIOS	525
DENTRO DEL PROCESO, EN LA MISMA MÁQUINA, REMOTO...	526
CARGA Y DESCARGA DE LA DLL	529
OLE Y EL REGISTRO DE WINDOWS	530
REGISTRANDO EL SERVIDOR	532
IMPLEMENTACIÓN DE INTERFACES	534
EL HUEVO, LA GALLINA Y LAS FÁBRICAS DE CLASES	536
IMPLEMENTANDO LA FÁBRICA DE CLASES	538
28. AUTOMATIZACIÓN OLE: CONTROLADORES	541
¿POR QUÉ EXISTE LA AUTOMATIZACIÓN OLE?	541
CONTROLADORES DE AUTOMATIZACIÓN CON VARIANTES	543
PROPIEDADES OLE Y PARÁMETROS POR NOMBRE	544
INTERFACES DUALES	545
BIBLIOTECAS DE TIPOS	546
IMPORTACIÓN DE BIBLIOTECAS DE TIPOS	547
EVENTOS	549
ESCUCHANDO A WORD	552
29. AUTOMATIZACIÓN OLE: SERVIDORES	557
INFORMES AUTOMATIZADOS	557
EL OBJETO DE AUTOMATIZACIÓN	559
LA PARTE CLIENTE	563
DECLARANDO UNA INTERFAZ COMÚN	564
MODELOS DE INSTANCIACIÓN	566
MODELOS DE CONCURRENCIA	568
UN SERVIDOR DE BLOQUEOS	570
LA IMPLEMENTACIÓN DE LA LISTA DE BLOQUEOS	572

CONTROL DE CONCURRENCIA	574
PONIENDO A PRUEBA EL SERVIDOR	577
30. MIDAS	579
¿QUÉ ES MIDAS?	579
CUÁNDO UTILIZAR Y CUÁNDO NO UTILIZAR MIDAS	581
MIDAS Y LAS BASES DE DATOS DE ESCRITORIO	583
MÓDULOS DE DATOS REMOTOS	584
PROVEEDORES	587
SERVIDORES REMOTOS Y CONJUNTOS DE DATOS CLIENTES	589
GRABACIÓN DE DATOS	591
RESOLUCIÓN	594
CONTROL DE ERRORES DURANTE LA RESOLUCIÓN	596
RECONCILIACIÓN	599
RELACIONES <i>MASTER/DETAIL</i> Y TABLAS ANIDADAS	601
ENVÍO DE PARÁMETROS	601
EXTENDIENDO LA INTERFAZ DEL SERVIDOR	602
ALGUIEN LLAMA A MI PUERTA	604
LA METÁFORA DEL MALETÍN	606
TIPOS DE CONEXIÓN	606
BALANCE DE CARGA SIMPLE	609
INTERFACES DUALES EN MIDAS	610
COGE EL DINERO Y CORRE: TRABAJO SIN CONEXIÓN	611
31. SERVIDORES DE INTERNET	617
EL MODELO DE INTERACCIÓN EN LA WEB	617
APRENDA HTML EN 14 MINUTOS	618
EXTENSIONES DEL SERVIDOR Y PÁGINAS DINÁMICAS	620
¿QUÉ NECESITO PARA ESTE SEGUIR LOS EJEMPLOS?	622
MÓDULOS WEB	623
ACCIONES	626
RECUPERACIÓN DE PARÁMETROS	628
GENERADORES DE CONTENIDO	629
GENERADORES DE TABLAS	631
MANTENIMIENTO DE LA INFORMACIÓN DE ESTADO	632
¿LE APETECE UNA GALLETA?	634
UN SIMPLE NAVEGADOR	635
AL OTRO LADO DE LA LÍNEA...	639
ACTIVEFORMS: FORMULARIOS EN LA WEB	640

LEFTOVERTURE

645

32. IMPRESIÓN DE INFORMES CON QUICKREPORT 647

LA HISTORIA DEL PRODUCTO	647
LA FILOSOFÍA DEL PRODUCTO	648
PLANTILLAS Y EXPERTOS PARA QUICKREPORT	649
EL CORAZÓN DE UN INFORME	650
LAS BANDAS	652
EL EVENTO <i>BEFOREPRINT</i>	654
COMPONENTES DE IMPRESIÓN	655
EL EVALUADOR DE EXPRESIONES	656
UTILIZANDO GRUPOS	657
ELIMINANDO DUPLICADOS	659
INFORMES <i>MASTER/DETAIL</i>	661
INFORMES COMPUESTOS	662
PREVISUALIZACIÓN A LA MEDIDA	663
LISTADOS AL VUELO	665
ENVIANDO CÓDIGOS BINARIOS A UNA IMPRESORA	667

33. ANÁLISIS GRÁFICO 671

GRÁFICOS Y BIORRITMOS	671
EL COMPONENTE <i>TDBCHART</i>	675
COMPONENTES NO VISUALES DE <i>DECISION CUBE</i>	677
REJILLAS Y GRÁFICOS DE DECISIÓN	679
USO Y ABUSO DE <i>DECISION CUBE</i>	681
MODIFICANDO EL MAPA DE DIMENSIONES	682

34. DESCENSO A LOS ABISMOS 685

INICIALIZACIÓN Y FINALIZACIÓN DEL BDE	685
EL CONTROL DE ERRORES	687
SESIONES Y CONEXIONES A BASES DE DATOS	688
CREACIÓN DE TABLAS	690
REESTRUCTURACIÓN	693
ELIMINACIÓN FÍSICA DE REGISTROS BORRADOS	695
CURSORES	696
UN EJEMPLO DE ITERACIÓN	698
PROPIEDADES	700
LAS FUNCIONES DE RESPUESTA DEL BDE	702

35. CREACIÓN DE INSTALACIONES 705

LOS PROYECTOS DE <i>INSTALLSHIELD EXPRESS</i>	705
LA PRESENTACIÓN DE LA INSTALACIÓN	707

LAS MACROS DE DIRECTORIOS	708
GRUPOS Y COMPONENTES	709
INSTALANDO EL BDE Y LOS SQL LINKS	711
CONFIGURACIÓN ADICIONAL DEL BDE	713
INSTALACIÓN DE PAQUETES	713
INTERACCIÓN CON EL USUARIO	714
LAS CLAVES DEL REGISTRO DE WINDOWS	716
CÓMO SE REGISTRAN LOS COMPONENTES ACTIVEX	717
ICONOS Y CARPETAS	718
GENERANDO Y PROBANDO LA INSTALACIÓN	719
LA VERSIÓN COMPLETA DE INSTALLSHIELD EXPRESS	720
LAS EXTENSIONES DE INSTALLSHIELD EXPRESS	721
36. EJEMPLOS: LIBRETAS DE AHORRO	723
DESCRIPCIÓN DEL MODELO DE DATOS	723
LIBRETAS DE AHORRO EN MS SQL SERVER	729
AHORA, EN ORACLE	733
EL MÓDULO DE DATOS	736
TRANSACCIONES EXPLÍCITAS	739
GESTIÓN DE LIBRETAS Y OPERACIONES	740
ENTRADA DE APUNTES	742
LA VENTANA PRINCIPAL	744
CORRIGIENDO EL IMPORTE DE UN APUNTE	746
37. EJEMPLOS: UN SERVIDOR DE INTERNET	749
BÚSQUEDA DE PRODUCTOS	749
EL MOTOR DE BÚSQUEDAS	751
CREANDO LA EXTENSIÓN WEB	754
GENERANDO LA TABLA DE RESULTADOS	756
DOCUMENTOS HTML Y SUSTITUCIÓN DE ETIQUETAS	757
RESPONDIENDO A LAS ACCIONES	759
APENDICE: EXCEPCIONES	761
SISTEMAS DE CONTROL DE ERRORES	761
CONTRATOS INCUMPLIDOS	762
CÓMO SE INDICA UN ERROR	763
LA EJECUCIÓN DEL PROGRAMA FLUYE EN DOS DIMENSIONES	764
PAGAMOS NUESTRAS DEUDAS	765
LA DESTRUCCIÓN DE OBJETOS DINÁMICOS	766
EL BLOQUE DE PROTECCIÓN DE RECURSOS	768
CÓMO TRANQUILIZAR A UN PROGRAMA ASUSTADO	770
EJEMPLOS DE CAPTURA DE EXCEPCIONES	771
CAPTURANDO EL OBJETO DE EXCEPCIÓN	772

16 *La Cara Oculta de C++ Builder*

CAPTURA Y PROPAGACIÓN DE EXCEPCIONES DE LA VCL	773
DISTINGUIR EL TIPO DE EXCEPCIÓN	773
LAS TRES REGLAS DE MARTEENS	774
CICLO DE MENSAJES Y MANEJO DE EXCEPCIONES	775
EXCEPCIONES A LA TERCERA REGLA DE MARTEENS	777
EL EVENTO <i>ONEXCEPTION</i>	778
LA EXCEPCIÓN SILENCIOSA	781
CONSTRUCTORES Y EXCEPCIONES	782
INDICE ALFABETICO	787

Prólogo del Autor

*“The day must come - if the world last long enough -” said Arthur, “when every possible tune will have been composed - every possible pun perpetrated ... and, worse than that, every possible book written! For the number of words is finite.” “It’ll make very little difference to the authors,” I suggested. “Instead of saying ‘what book shall I write?’ an author will ask himself ‘which book shall I write?’ A mere verbal distinction!” Lady Muriel gave me an approving smile. “But lunatics would always write new books, surely?” she went on. “They couldn’t write the same books over again!”
Lewis Carroll - Sylvie and Bruno Concluded*

NO CREO EN LAS ENCICLOPEDIAS; de hecho, nunca he podido terminar de leer alguna. Sin embargo, cada vez que hojeo un nuevo libro sobre C++ Builder o Delphi me da la impresión de que el autor ha intentado precisamente escribir una enciclopedia; lástima que, en la mayoría de los casos, el volumen del libro se limita a lo sumo a un tomo de no más de 1000 páginas.

Este libro no pretende ni puede abarcar todos los temas relacionados con la programación en C++ Builder; su objetivo concreto son LAS TÉCNICAS DE PROGRAMACIÓN DE BASES DE DATOS UTILIZANDO C++ BUILDER, con énfasis especial en el desarrollo de aplicaciones cliente/servidor y de múltiples capas, tarea para la que los entornos de programación de Borland están particularmente bien preparados.

Por lo tanto, asumiré cierta familiaridad del lector con C++, bastante común entre los programadores de hoy día; sería impensable intentar suplantar a clásicos de tal envergadura como “*El Lenguaje de Programación C++*” del creador del lenguaje, Bjarne Stroustrup.

Contenido del libro

El libro se divide en cinco partes, que son las siguientes:

- La **Parte 1** – “El lenguaje SQL” – ofrece una extensa presentación de los elementos de SQL, tanto en lo referente a las instrucciones básicas del estándar ANSI 92, como en todo lo relacionado con la programación del lado del servidor: el lenguaje de *triggers* y procedimientos almacenados. En este último aspecto, los conceptos se presentan utilizando la sintaxis de InterBase, el servidor SQL de Borland. Posteriormente, presentamos por separado las características distintivas de los tres sistemas cliente/servidor más utilizados hoy: Microsoft SQL Server, IBM DB2 Universal Database y Oracle.

- La **Parte 2** – “Navegación y búsqueda” – describe las facilidades básicas que ofrecen los componentes VCL incluidos en C++ Builder para que representemos los conjuntos de datos con los que vamos a trabajar y nos desplazemos por los registros que los componen. Esta parte se encarga también de presentar la filosofía de utilización de controles visuales con conexión a datos (*data-aware controls*) y de mostrar ejemplos de cómo obtener interfaces de usuario atractivas para mostrar la información. Para concluir esta parte, “*Comunicación cliente/servidor*” describe las técnicas generales que deben tenerse en cuenta al interactuar con bases de datos remotas; en este capítulo mostramos la inconsistencia de varios mitos de amplia difusión entre la comunidad de programadores.
- En la **Parte 3** – “Actualizaciones y concurrencia” – se describen en detalle las posibilidades que ofrece C++ Builder a la hora de actualizar el contenido de nuestras bases de datos. En estos capítulos se hace especial énfasis en todo lo relacionado con la optimización del *acceso concurrente* a las bases de datos, de forma que sea posible minimizar la contención entre aplicaciones que se ejecuten simultáneamente desde diferentes clientes, y se describen las posibilidades que ofrece el Motor de Datos de Borland (*Borland Database Engine - BDE*) para la implementación de transacciones y las ventajas y limitaciones en este sentido de los diferentes sistemas SQL con los que el BDE puede comunicarse.
- La **Parte 4** – “Programación Distribuida” – es, técnicamente hablando, la más compleja del libro. En esta parte describiré inicialmente en detalle las posibilidades del componente *TClientDataSet*, la clase sobre la que se basa la programación de *clientes delegados* para el desarrollo de aplicaciones de múltiples capas en C++ Builder. Luego introduciré los conceptos fundamentales del Modelo de Objetos Componentes (COM) de Microsoft, la base sobre la que se apoya la programación de servidores de capa intermedia; inmediatamente después, un amplio capítulo describe en detalle las posibilidades de la tecnología que Borland ha llamado MIDAS y se presentan ejemplos de aplicaciones de tres capas. Pero todavía le quedará mucho más por ver en esta parte: el siguiente capítulo muestra cómo utilizar los asistentes y componentes que incluye C++ Builder para crear extensiones de servidores de Internet que construyan dinámicamente páginas HTML a partir del contenido de bases de datos.
- En la **Parte 5**, que he denominado “Leftoverture”¹, se aglutina toda una serie de capítulos que en principio no tienen una conexión directa entre sí. Aquí encontrará todo lo relacionado con cómo utilizar *QuickReport*, la herramienta de generación de informes que se incluye en C++ Builder, cómo incluir en sus aplicaciones tablas de análisis multidimensional de datos a partir de los componentes *Decision Cube*, o cómo generar instalaciones con *InstallShield Express for C++*

¹ “Leftoverture” es el nombre de un álbum producido en 1976 por Kansas. El nombre es una combinación de las palabras “leftover” (remanente, residuo) y “overture” (obertura). La pieza clave del álbum fue compuesta mezclando trozos de otras canciones que habían sido descartadas por el grupo.

Builder. Los dos últimos capítulos presentan dos ejemplos completos de aplicaciones, cuyo código fuente podrá utilizar como material de estudio.

Mi propósito es que éste sea un libro vivo, que crezca a la vista de todos. Así intento evitar la incómoda sensación que queda al entregar los ficheros a la imprenta de que te has olvidado de mencionar algo importante. ¿Cómo crece este libro? Visite mi página Web, en www.marteens.com, donde podrá encontrar trucos y artículos técnicos que sirven de extensión a estas páginas.

Agradecimientos

Como siempre, los protagonistas de todos mis libros son todos los programadores que tengo el gusto de conocer y trabajar en alguna medida con ellos. Lo poco de original que hay en estas páginas casi siempre ha surgido de problemas reales que nos hemos vistos obligados a enfrentar en conjunto. A veces la solución ha sido nuestra, pero muchas otras nos han salvado la vida tantas personas que nos sería imposible intentar mencionarlas a todas. Y están los cientos de programadores que, a través del correo electrónico, nos han animado a seguir adelante. Pero la principal fuerza que ha movido este proyecto ha sido Dave. Sin él, este libro no estaría ahora en sus manos.

Como no me gusta ponerme solemne, quiero concluir agradeciendo a Dios, al Diablo o a la Selección Natural (no estoy seguro), por haberme colocado en el cráneo esa neurona sarcástica que impide que tome en serio a algo o a alguien, incluyéndome a mí mismo. El día que me falle ... quizás me dedique a escribir sobre la capa de ozono.

Ian Marteens
Madrid, Junio de 1999

1

El Lenguaje SQL

- **Sistemas de bases de datos**
- **Breve introducción a SQL**
- **Consultas y modificaciones**
- **Procedimientos almacenados y triggers**
- **Transacciones**
- **Microsoft SQL Server**
- **Oracle**
- **DB2 Universal Database**
- **El Motor de Datos de Borland**

Parte

Sistemas de bases de datos

ESTE ES EL MOMENTO APROPIADO para presentar a los protagonistas de este drama: los sistemas de gestión de bases de datos con los que intentaremos trabajar. En mi trabajo de consultor, la primera pregunta que escucho, y la más frecuente, es en qué lenguaje debe realizarse determinado proyecto. Enfoque equivocado. La mayoría de los proyectos que llegan a mis manos son aplicaciones para redes de área local, y os garantizo que la elección del lenguaje es asunto relativamente secundario para el éxito de las mismas; por supuesto, siempre recomiendo algún lenguaje “decente”, sin limitaciones intrínsecas, como C++ Builder o Delphi. La primera pregunta debería ser: ¿qué sistema de bases de datos es el más apropiado a mis necesidades?

En este capítulo recordaremos los principios básicos de los sistemas de bases de datos relacionales, y haremos una rápida introducción a algunos sistemas concretos con los que C++ Builder puede trabajar de modo directo. La explicación relativa a los sistemas de bases de datos locales será más detallada que la de los sistemas cliente/servidor, pues las características de estos últimos (implementación de índices, integridad referencial, seguridad) irán siendo desveladas a lo largo del libro.

Acerca del acceso transparente a bases de datos

¿Pero acaso C++ Builder no ofrece cierta transparencia con respecto al formato de los datos con los que estamos trabajando? Pues sí, sobre todo para los formatos soportados por el Motor de Bases de Datos de Borland (BDE), que estudiaremos en el capítulo correspondiente. Sin embargo, esta transparencia no es total, incluso dentro de las bases de datos accesibles mediante el BDE. Hay una primera gran división entre las bases de datos locales y los denominados sistemas SQL. Luego vienen las distintas posibilidades expresivas de los formatos; incluso las bases de datos SQL, que son las más parecidas entre sí, se diferencian en las operaciones que permiten o no. Si usted está dispuesto a utilizar el mínimo común denominador entre todas ellas, su aplicación puede que funcione sin incidentes sobre determinado rango de posibles sistemas ... pero le aseguro que del mismo modo desperdiciará recursos de progra-

mación y diseño que le habrían permitido terminar mucho antes la aplicación, y que ésta se ejecutara más eficientemente.

Bases de datos relacionales

Por supuesto, estimado amigo, todos los sistemas de bases de datos con los que vamos a trabajar en C++ Builder serán sistemas relacionales. Por desgracia. ¿Cómo que por desgracia, hereje insensato? Para saber qué nos estamos perdiendo con los sistemas relacionales tendríamos que conocer las alternativas. Necesitaremos, lo lamento, un poco de vieja Historia.

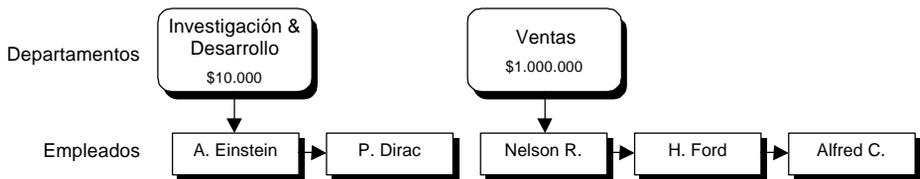
En el principio no había ordenadores, claro está. Pero cuando los hubo, y después de largos años de almacenar información “plana” en grandes cintas magnéticas o perforadas, los informáticos comenzaron a organizar sus datos en dos tipos de modelos: el modelo jerárquico y el modelo de redes. El modelo jerárquico era un invento digno de un oficial prusiano. Los diferentes tipos de información se clasificaban en forma de árbol. En determinada base de datos, por ejemplo, la raíz de este árbol eran los registros de empresas. Cada empresa almacenaría los datos de un conjunto de departamentos, estos últimos serían responsables de guardar los datos de sus empleados, y así sucesivamente. Además del conjunto de departamentos, una empresa podría ser propietaria de otro conjunto de registros, como bienes inmuebles o algo así. El problema, como es fácil de imaginar, es que el mundo real no se adapta fácilmente a este tipo de organización. Por lo tanto, a este modelo de datos se le añaden chapuzas tales como “registros virtuales” que son, en el fondo, una forma primitiva de punteros entre registros.

El modelo de redes era más flexible. Un registro podía contener un conjunto de otros registros. Y cada uno de estos registros podía pertenecer a más de un conjunto. La implementación más frecuente de esta característica se realizaba mediante punteros. El sistema más famoso que seguía este modelo, curiosamente, fue comprado por cierta compañía tristemente conocida por hacerse con sistemas de bases de datos para convertirlos en historia... No, no es esa Compañía en la que estáis pensando... sí, esa Otra...

Vale, el modelo jerárquico no era muy completo, pero el modelo de redes era razonablemente bueno. ¿Qué pasó con ellos, entonces? ¿Por qué se extinguieron en el Jurásico? Básicamente, porque eran sistemas *navegacionales*. Para obtener cualquier información había que tener una idea muy clara de cómo estaban organizados los datos. Pero lo más molesto era que no existían herramientas sencillas que permitieran realizar consultas arbitrarias en una base de datos. Si el presidente de la compañía quería saber cuántos clientes del área del Pacífico bebían Coca-Cola a las cinco de la tarde en vez de té, tenía que llamar al programador para que le desarrollara una pequeña aplicación.

Entonces apareció Mr. Codd, un matemático de IBM. No inventó el concepto de registro, que ya existía hacía tiempo. Pero se dio cuenta que si obligaba a que todos los campos de los registros fueran campos simples (es decir, que no fueran punteros, vectores o subregistros) podía diseñarse un grácil sistema matemático que permitía descomponer información acerca de objetos complejos en estos registros planos, con la seguridad de poder restaurar la información original más adelante, con la ayuda de operaciones algebraicas. Lo más importante: casi cualquier tipo de información podía descomponerse de este modo, así que el modelo era lo suficientemente general. A la teoría matemática que desarrolló se le conoce con el nombre de *álgebra relacional*, y es la base de nuestro conocido lenguaje SQL y del quizás menos popular *Query By Example*, o QBE. De este modo, el directivo del párrafo anterior podía sentarse frente a una consola, teclear un par de instrucciones en SQL y ahorrarse el pago de las horas extras del programador².

Tomemos como ejemplo una base de datos que almacene datos acerca de Departamentos y sus Empleados. Los modelos jerárquicos y de redes la representarían con un diagrama similar al siguiente:



Como puede verse, los punteros son parte ineludible del modelo. ¿Hay algún matemático que sepa cómo comportarse frente a un puntero? Al parecer, no los había en los 60 y 70. Sin embargo, los datos anteriores pueden expresarse, en el modelo relacional, mediante dos conjuntos uniformes de registros y sin utilizar punteros, al menos de forma explícita. De esta manera, es factible analizar matemáticamente los datos, y efectuar operaciones algebraicas sobre los mismos:

DEPARTAMENTOS		
Codigo	Nombre	Presupuesto
I+D	Investigación y Desarrollo	\$10.000
V	Ventas	\$1.000.000

EMPLEADOS	
Dpto	Nombre
I+D	A. Einstein
I+D	P. Dirac
V	Nelson R.
V	Henri F.
V	Alfred C.

¡Mira, mamá, sin punteros! *Departamentos* y *Empleados* son *tablas*, aunque matemáticamente se les denomina *relaciones*. A los registros de estas tablas se les llama *filas*, para

² Aunque personalmente no conozco a ningún individuo con alfiler de corbata y BMW que sepa SQL, no cabe duda de que debe existir alguno por ahí.

hacer rabiar a los matemáticos que les llaman *tuplas*. Y para no desentonar, *Código*, *Nombre* y *Presupuesto* son *columnas* para unos, mientras que para los otros son *campos*. ¡Qué más da! Ah, la colección de tablas o relaciones es lo que se conoce como *base de datos*.

Las personas inteligentes (como usted y como yo) se dan cuenta enseguida de que en realidad no hemos eliminado los punteros, sino que los hemos disfrazado. Existe un vínculo³ entre los campos *Código* y *Dpto* que es el sustituto de los punteros. Pero cuando se representan los datos de esta manera, es más fácil operar con ellos matemáticamente. Note, por ejemplo, que para ilustrar los modelos anteriores necesité un dibujo, mientras que una vulgar tabla de mi procesador de texto me ha bastado en el segundo caso. Bueno, en realidad han sido dos tablas.

¿Me deja el lector que resuma en un par de frases lo que lograba Codd con su modelo relacional? Codd apuntaba su pistola de rayos desintegradores a cualquier objeto que se ponía a tiro, incluyendo a su perro, y lo reducía a cenizas atómicas. Después, con un elegante par de operaciones matemáticas, podía resucitar al animalito, si antes el viento no barría sus restos de la alfombra.

Información semántica = restricciones

Todo lo que he escrito antes le puede sonar al lector como un disco rayado de tanto escucharlo. Sin embargo, gran parte de los programadores que se inician en C++ Builder solamente han llegado a asimilar esta parte básica del modelo relacional, y presentan lagunas aterradoras en el resto de las características del modelo, como veremos dentro de poco. ¿Qué le falta a las ideas anteriores para que sean completamente prácticas y funcionales? Esencialmente, información semántica: algo que nos impida o haga improbable colocar la cabeza del perro donde va la cola, o viceversa (el perro de una vecina mía da esa impresión).

Casi siempre, esta información semántica se expresa mediante restricciones a los valores que pueden tomar los datos de una tabla. Las restricciones más sencillas tienen que ver con el tipo de valores que puede albergar una columna. Por ejemplo, la columna *Presupuesto* solamente admite valores enteros. Pero no cualquier valor entero: tienen que ser valores positivos. A este tipo de verificaciones se les conoce como *restricciones de dominio*.

El nivel siguiente lo conforman las restricciones que pueden verificarse analizando los valores de cada fila de forma independiente. Estas no tienen un nombre especial.

³ Observe con qué exquisito cuidado he evitado aquí la palabra *relación*. En inglés existen dos palabras diferentes: *relation* y *relationship*. Pero el equivalente más cercano a esta última sería algo así como *relacionalidad*, y eso suena peor que un párrafo del BOE.

En el ejemplo de los departamentos y los empleados, tal como lo hemos presentado, no hay restricciones de este tipo. Pero nos podemos inventar una, que deben satisfacer los registros de empleados:

Dpto <> "I+D" or Especialidad <> "Psiquiatría"

Es decir, que no pueden trabajar psiquiatras en Investigación y Desarrollo (terminarían igual de locos). Lo más importante de todo lo que he explicado en esta sección es que las restricciones más sencillas pueden expresarse mediante elegantes fórmulas matemáticas que utilizan los nombres de las columnas, o campos, como variables.

Restricciones de unicidad y claves primarias

Los tipos de restricciones siguen complicándose. Ahora se trata de realizar verificaciones que afectan los valores almacenados en varias filas. Las más importantes de estas validaciones son las denominadas *restricciones de unicidad*. Son muy fáciles de explicar en la teoría. Por ejemplo, no pueden haber dos filas en la tabla de departamentos con el mismo valor en la columna *Codigo*. Abusando del lenguaje, se dice que “la columna *Codigo* es única”.

En el caso de la tabla de departamentos, resulta que también existe una restricción de unicidad sobre la columna *Nombre*. Y no pasa nada. Sin embargo, quiero que el lector pueda distinguir sin problemas entre estas dos diferentes situaciones:

1. (Situación real) Hay una restricción de unicidad sobre *Codigo* y otra restricción de unicidad sobre *Nombre*.
2. (Situación ficticia) Hay una restricción de unicidad sobre la combinación de columnas *Codigo* y *Nombre*.

Esta segunda restricción posible es más relajada que la combinación real de dos restricciones (compruébelo). La unicidad de una combinación de columnas puede visualizarse de manera sencilla: si se “recortan” de la tabla las columnas que no participan en la restricción, no deben quedar registros duplicados después de esta operación. Por ejemplo, en la tabla de empleados, la combinación *Dpto* y *Nombre* es única.

La mayoría de los sistemas de bases de datos se apoyan en índices para hacer cumplir las restricciones de unicidad. Estos índices se crean automáticamente tomando como base a la columna o combinación de columnas que deben satisfacer estas condiciones. Antes de insertar un nuevo registro, y antes de modificar una columna de este tipo, se busca dentro del índice correspondiente para ver si se va a generar un valor duplicado. En tal caso se aborta la operación.

Así que una tabla puede tener una o varias restricciones de unicidad (o ninguna). Escoja una de ellas y designela como *clave primaria*. ¿Cómo, de forma arbitraria? Casi: se supone que la clave primaria identifica unívocamente y de forma algo misteriosa la más recóndita esencia de los registros de una tabla. Pero para esto vale lo mismo cualquier otra restricción de unicidad, y en programación no vale recurrir al misticismo. Hay quienes justifican la elección de la clave primaria entre todas las restricciones de unicidad en relación con la integridad referencial, que estudiaremos en la próxima sección. Esto tampoco es una justificación, como veremos en breve. En definitiva, que todas las restricciones de unicidad son iguales, aunque algunas son más iguales que otras.

¿Quiere saber la verdad? He jugado con trampa en el párrafo anterior. Esa esencia misteriosa del registro no es más que un mecanismo utilizado por el modelo relacional para sustituir a los desterrados punteros. Podrís llamarlo *identidad del registro*, o cualquier otro nombre rimbombante, pero no es más que una forma de disfrazar un puntero, y muy poco eficiente, por cierto. Observe la tabla de departamentos: entre el código y el nombre, ¿cuál columna elegiría como clave primaria? Por supuesto que el código, pues es el tipo de datos que menos espacio ocupa, y cuando tengamos el código en la mano podremos localizar el registro de forma más rápida que cuando tengamos el nombre.

De todos modos, hay alguien que aprovecha inteligentemente la existencia de claves primarias: el Motor de Datos de Borland. De hecho, estas claves primarias son la forma en que el Motor de Datos puede simular el concepto de registro activo en una tabla SQL. Pero tendremos que esperar un poco para desentrañar este ingenioso mecanismo.

Integridad referencial

Este tipo de restricción es aún más complicada, y se presenta en la columna *Dpto* de la tabla de empleados. Es evidente que todos los valores de esta columna deben corresponder a valores almacenados en la columna *Código* de la tabla de departamentos. Siguiendo mi teoría de la conspiración de los punteros encubiertos, esto equivale a que cada registro de empleado tenga un puntero a un registro de departamento.

Un poco de terminología: a estas restricciones se les denomina *integridad referencial*, o *referential integrity*. También se dice que la columna *Dpto* de *Empleados* es una *clave externa* de esta tabla, o *foreign key*, en el idioma de Henry Morgan. Podemos llamar tabla dependiente, o tabla de detalles, a la tabla que contiene la clave externa, y tabla maestra a la otra. En el ejemplo que consideramos, la clave externa consiste en una sola columna, pero en el caso general podemos tener claves externas compuestas.

Como es fácil de ver, las columnas de la tabla maestra a las que se hace referencia en una integridad referencial deben satisfacer una restricción de unicidad. En la teoría original, estas columnas deberían ser la clave primaria, pero la mayoría de los sistemas relacionales actuales admiten cualquiera de las combinaciones de columnas únicas definidas.

Cuando se establece una relación de integridad referencial, la tabla dependiente asume responsabilidades:

- No se puede insertar una nueva fila con un valor en las columnas de la clave externa que no se encuentre en la tabla maestra.
- No se puede modificar la clave externa en una fila existente con un valor que no exista en la tabla maestra.

Pero también la tabla maestra tiene su parte de responsabilidad en el contrato, que se manifiesta cuando alguien intenta eliminar una de sus filas, o modificar el valor de su clave primaria. En las modificaciones, en general, pueden desearse dos tipos diferentes de comportamiento:

- Se prohíbe la modificación de la clave primaria de un registro que tenga filas de detalles asociadas.
- Alternativamente, la modificación de la clave se propaga a las tablas dependientes.

Si se trata de un borrado, son tres los comportamientos posibles:

- Prohibir el borrado, si existen filas dependientes del registro.
- Borrar también las filas dependientes.
- Permitir el borrado y, en vez de borrar las filas dependientes, romper el vínculo asociando a la clave externa un valor por omisión, que en SQL casi siempre es el valor *nulo* (ver el siguiente capítulo).

La forma más directa de implementar las verificaciones de integridad referencial es utilizar índices. Las responsabilidades de la tabla dependiente se resuelven comprobando la existencia del valor a insertar o a modificar en el índice asociado a la restricción de unicidad definida en la tabla maestra. Las responsabilidades de la tabla maestra se resuelven generalmente mediante índices definidos sobre la tabla de detalles.

¿Qué tiene de malo el modelo relacional?

El modelo relacional funciona. Además, funciona razonablemente bien. Pero como he tratado de explicar a lo largo de las secciones anteriores, en determinados aspectos

significa un retroceso en comparación con modelos de datos anteriores. Me refiero a lo artificioso del proceso de eliminar los punteros implícitos de forma natural en el modelo semántico a representar. Esto se refleja también en la eficiencia de las operaciones. Supongamos que tenemos un registro de empleado en nuestras manos y queremos saber el nombre del departamento al que pertenece. Bien, pues tenemos que buscar el código de departamento dentro de un índice para después localizar físicamente el registro del departamento correspondiente y leer su nombre. Al menos, un par de accesos al disco duro. Compare con la sencillez de buscar directamente el registro de departamento dado su puntero (existen implementaciones eficientes del concepto de puntero cuando se trabaja con datos persistentes).

En este momento, las investigaciones de vanguardia se centran en las bases de datos orientadas a objetos, que retoman algunos conceptos del modelo de redes y del propio modelo relacional. Desde la década de los 70, la investigación matemática ha avanzado lo suficiente como para disponer de potentes lenguajes de interrogación sobre bases de datos arbitrarias. No quiero entrar a analizar el porqué no se ha acabado de imponer el modelo orientado a objetos sobre el relacional, pues en esto influyen tanto factores técnicos como comerciales. Pero es bueno que el lector sepa qué puede pasar en un futuro cercano.

Es sumamente significativo que la principal ventaja del modelo relacional, la posibilidad de realizar consultas *ad hoc* estuviera fuera del alcance del modelo “relacional” más popular a lo largo de los ochenta: dBase, y sus secuelas Clipper y FoxPro. Cuando escucho elogios sobre Clipper por parte de programadores que hicieron carrera con este lenguaje, pienso con tristeza que los elogios los merecen los propios programadores que pudieron realizar software funcional con herramientas tan primitivas. Mi aplauso para ellos; en ningún caso para el lenguaje. Y mi consejo de que abandonen el viejo buque (y las malas costumbres aprendidas durante la travesía) antes de que termine de hundirse.



Ilustración 1 El perro de Codd

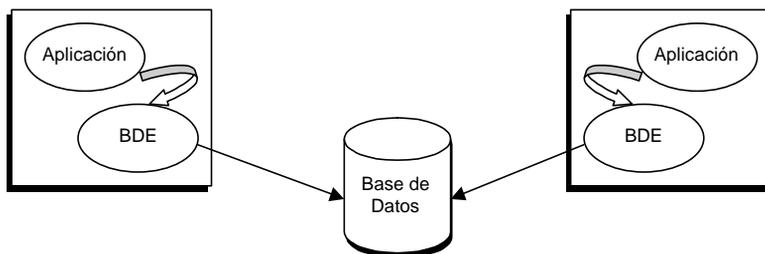
Y a todas estas, ¿qué pasó con la mascota de Mr. Codd? Lo inevitable: murió como consecuencia de un experimento fallido. Sin embargo, Brahma se apiadó de él, y reencarnó al año en un chico que, con el tiempo, se ha convertido en un exitoso programador afincado en el sur de la Florida. Al verlo, nadie pensaría que fue un perro en su vida anterior, de no ser por la manía que tiene de rascarse periódicamente la oreja. No obstante, por haber destrozado la tapicería del sofá de su dueño y orinarse un par de veces en la alfombra del salón, fue condenado a programar dos largos años en Visual Basic, hasta que C++ Builder (¿o fue Delphi?) llegó a su vida y se convirtió en un hombre feliz.

Bases de datos locales y servidores SQL

Basta ya de teoría, y veamos los ingredientes con que contamos para cocinar aplicaciones de bases de datos. La primera gran división entre los sistemas de bases de datos existentes se produce entre los sistemas locales, o de escritorio, y las bases de datos SQL, o cliente/servidor.

A los sistemas de bases de datos locales se les llama de este modo porque comenzaron su existencia como soluciones baratas para un solo usuario, ejecutándose en un solo ordenador. Sin embargo, no es un nombre muy apropiado, porque más adelante estos sistemas crecieron para permitir su explotación en red. Tampoco es adecuado clasificarlas como “lo que queda después de quitar las bases de datos SQL”. Es cierto que en sus inicios ninguna de estas bases de datos soportaba un lenguaje de consultas decente, pero esta situación también ha cambiado.

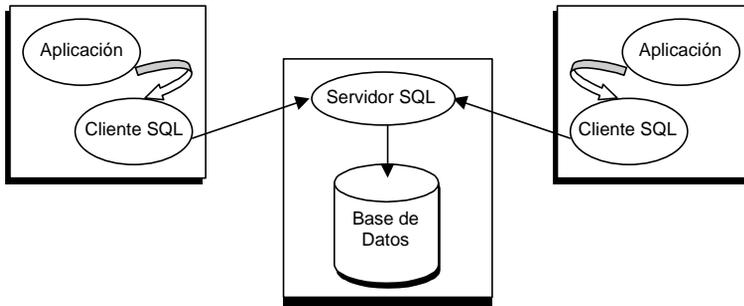
En definitiva, ¿cuál es la esencia de las bases de datos de escritorio? Pues el hecho de que la programación usual con las mismas se realiza en una sola capa. Todos estos sistemas utilizan como interfaz de aplicaciones un motor de datos que, en la era de la supremacía de Windows, se implementa como una DLL. En la época de MS-DOS, en cambio, eran funciones que se enlazaban estáticamente dentro del ejecutable. Observe el siguiente diagrama, que representa el uso en red de una base de datos “de escritorio”:



Aunque la segunda burbuja dice “BDE”, el Motor de Datos de Borland, sustituya estas siglas por DAO, y podrá aplicar el diagrama a Access. He representado la aplicación y el motor de datos en dos burbujas separadas, pero en realidad, constituyen un mismo ejecutable. Lo más importante, sin embargo, es que no existe un software central que sirva de árbitro para el acceso a la base de datos “física”. Es como si las dos aplicaciones deambularan a ciegas en un cuarto oscuro, tratando de sentarse en algún sitio libre. Por supuesto, la única forma que tienen de saberlo es intentar hacerlo y confiar en que no haya nadie debajo.

Debido a esta forma primitiva de resolver las inevitables colisiones, la implementación de la concurrencia, las transacciones y, en último término, la recuperación después de fallos, ha sido tradicionalmente el punto débil de las bases de datos de escritorio. Si está pensando en decenas o cientos de usuarios atacando simultáneamente a sus datos, y en ejércitos de extremidades inferiores listas para tropezar con cables, olvídense de Paradox, dBase y Access, pues necesitará una base de datos cliente/servidor.

Las bases de datos cliente/servidor, o bases de datos SQL, se caracterizan por utilizar al menos dos capas de software, como se aprecia en el siguiente diagrama:



El par aplicación + motor local de datos ya no tiene acceso directo a los ficheros de la base de datos, pues hay un nuevo actor en el drama: el servidor SQL. He cambiado el rótulo de la burbuja del motor de datos, y ahora dice “cliente SQL”. Esta es una denominación genérica. Para las aplicaciones desarrolladas con C++ Builder y el Motor de Datos de Borland, este cliente consiste en la combinación del BDE propiamente dicho *más* alguna biblioteca dinámica o DLL suministrada por el fabricante de la base de datos. En cualquier caso, todas estas bibliotecas se funden junto a la aplicación dentro de una misma capa de software, compartiendo el mismo espacio de memoria y procesamiento.

La división entre bases de datos de escritorio y las bases de datos SQL no es una clasificación tajante, pues se basa en la combinación de una serie de características.

Puede que uno de estos días aparezca un sistema que mezcle de otra forma estos rasgos definitorios.

Características generales de los sistemas SQL

El hecho de que exista un árbitro en las aplicaciones cliente/servidor hace posible implementar una gran variedad de técnicas y recursos que están ausentes en la mayoría de los sistemas de bases de datos de escritorio. Por ejemplo, el control de concurrencia se hace más sencillo y fiable, pues el servidor puede llevar la cuenta de qué clientes están accediendo a qué registros durante todo el tiempo. También es más fácil implementar transacciones atómicas, esto es, agrupar operaciones de modificación de forma tal que, o se efectúen todas, o ninguna llegue a tener efecto.

Pero una de las principales características de las bases de datos con las que vamos a trabajar es la forma peculiar en que “conversan” los clientes con el servidor. Resulta que estas conversaciones tienen lugar en forma de petición de ejecución de comandos del lenguaje SQL. De aquí el nombre común que reciben estos sistemas. Supongamos que un ordenador necesita leer la tabla de inventario. Recuerde que ahora no podemos abrir directamente un fichero situado en el servidor. Lo que realmente hace la estación de trabajo es pedirle al servidor que ejecute la siguiente instrucción SQL:

```
select * from Inventario order byCodigoProducto asc
```

El servidor calcula qué registros pertenecen al resultado de la consulta, y todo este cálculo tiene lugar en la máquina que alberga al propio servidor. Entonces, cada vez que el usuario de la estación de trabajo se mueve de registro a registro, el cliente SQL pide a su servidor el siguiente registro mediante la siguiente instrucción:

```
fetch
```

Más adelante necesitaremos estudiar cómo se realizan las modificaciones, inserciones, borrados y búsquedas, pero con esto basta por el momento. Hay una conclusión importante a la que ya podemos llegar: ¿qué es más rápido, pulsar un botón en la máquina de bebidas o mandar a un acólito a que vaya por una Coca-Cola? A no ser que usted sea más vago que yo, preferirá la primera opción. Bien, pues un sistema SQL es inherentemente más lento que una base de datos local. Antes manipulábamos directamente un fichero. Ahora tenemos que pedirselo a alguien, con un protocolo y con reglas de cortesía. Ese alguien tiene que entender nuestra petición, es decir, compilar la instrucción. Luego debe ejecutarla y solamente entonces procederá a enviarnos el primer registro a través de la red. ¿Está de acuerdo conmigo?

No pasa una semana sin que conozca a alguien que ha desarrollado una aplicación para bases de datos locales, haya decidido pasarla a un entorno SQL, y haya pensado

que con un par de modificaciones en su aplicación era suficiente. Entonces descubre que la aplicación se ejecuta con mayor lentitud que antes, cae de rodillas, mira al cielo y clama: ¿dónde está entonces la ventaja de trabajar con sistemas cliente/servidor? Está, amigo mío, en la posibilidad de meter código en el servidor, y si fallamos en hacerlo estaremos desaprovechando las mejores dotes de nuestra base de datos.

Hay dos formas principales de hacerlo: mediante procedimientos almacenados y mediante *triggers*. Los primeros son conjuntos de instrucciones que se almacenan dentro de la propia base de datos. Se activan mediante una petición explícita de un cliente, pero se ejecutan en el espacio de aplicación del servidor. Por descontado, estos procedimientos no deben incluir instrucciones de entrada y salida. Cualquier proceso en lote que no contenga este tipo de instrucciones es candidato a codificarse como un procedimiento almacenado. ¿La ventaja?, que evitamos que los registros procesados por el procedimiento tengan que atravesar la barrera del espacio de memoria del servidor y viajar a través de la red hasta el cliente.

Los *triggers* son también secuencias de instrucciones, pero en vez de ser activados explícitamente, se ejecutan como preludio y coda de las tres operaciones básicas de actualización de SQL: **update**, **insert** y **delete**. No importa la forma en que estas tres operaciones se ejecuten, si es a instancias de una aplicación o mediante alguna herramienta incluida en el propio sistema; los *triggers* que se hayan programado se activarán en cualquier caso.

Estos recursos se estudiarán más adelante, cuando exploremos el lenguaje SQL.

El formato Paradox

Comenzaremos nuestra aventura explicando algunas características de los sistemas de bases de datos de escritorio que pueden utilizarse directamente con C++ Builder. De estos, mi actual favorito es Paradox. Pongo el adjetivo “actual” porque el nuevo formato de Visual dBase VII comienza a aproximarse a la riqueza expresiva soportada desde hace mucho por Paradox. Dedicaremos a dBase la siguiente sección.

Paradox no reconoce directamente el concepto de *base de datos*, sino que administra tablas en ficheros independientes. Cada tabla se almacena en un conjunto de ficheros, todos con el mismo nombre, pero con las extensiones que explicamos a continuación:

Extensión	Explicación
.db	Definición de la tabla y campos de longitud máxima fija
.mb	Campos de longitud variable, como los memos y gráficos
.px	El índice de la clave primaria

Extensión	Explicación
<i>.Xnn, .Ynn</i>	Indices secundarios
<i>.val</i>	Validaciones e integridad referencial

En el fichero *.db* se almacena una cabecera con la descripción de la tabla, además de los datos de los registros que corresponden a campos de longitud fija. Este fichero está estructurado en bloques de idéntico tamaño; se pueden definir bloques de 1, 2, 4, 8, 16 y 32KB. El tamaño de bloque se determina durante la creación de la tabla mediante el parámetro *BLOCK SIZE* del controlador de Paradox del BDE (por omisión, 2048 bytes); en el capítulo 16 aprenderemos a gestionar éste y muchos otros parámetros. El tamaño de bloque influye en la extensión máxima de la tabla, pues Paradox solamente permite 2^{16} bloques por fichero. Si se utiliza el tamaño de bloque por omisión tendríamos el siguiente tamaño máximo por fichero de datos:

$$2048 \times 65536 = 2^{11} \times 2^{16} = 2^{27} = 2^7 \times 2^{20} = 128\text{MB}$$

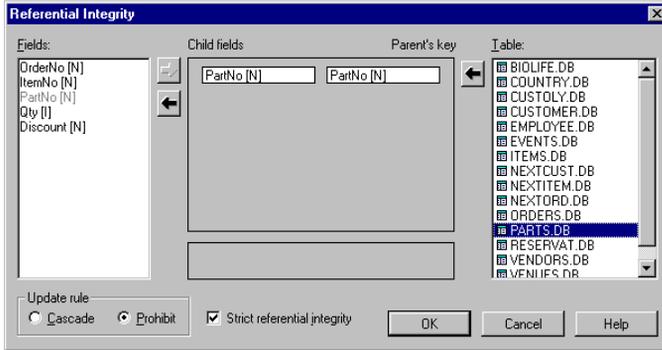
Un registro debe caber completamente dentro de un bloque, y si la tabla tiene una clave primaria (¡recomendable!) deben existir al menos tres registros por bloque; esto se refiere a los datos de longitud fija del registro, excluyendo campos de texto largos, de imágenes y binarios. Dentro de cada bloque, los registros se ordenan según su clave primaria, para agilizar la búsqueda una vez que el bloque ha sido leído en la caché. Un registro, además, siempre ocupa el mismo espacio dentro del fichero *.db*, incluso si contiene campos alfanuméricos que, a diferencia de lo que sucede en dBase, no se rellenan con blancos hasta ocupar el ancho total del campo.

Paradox permite trabajar con un índice primario por cada tabla y con varios índices secundarios. Todos estos índices pueden ser mantenidos automáticamente por el sistema, aunque existe la posibilidad de crear índices secundarios no mantenidos. El índice primario de la tabla se almacena en el fichero de extensión *.px*. No es necesario que una tabla tenga índice primario, pero es altamente ventajoso. Al estar ordenados los registros dentro los bloques de datos, sólo se necesita almacenar en el índice la clave del primer registro de cada bloque. Esto disminuye considerablemente los requerimientos de espacio del índice, y acelera las búsquedas.

Es interesante conocer cómo Paradox implementa los índices secundarios. En realidad, cada índice secundario es internamente una tabla, con su propio fichero de datos, de extensión *.Xnn*, y su índice asociado *.Ynn*. La numeración de los índices sigue un esquema sencillo: los dos últimos caracteres de la extensión indican el número del campo en hexadecimal, si es un índice sobre un solo campo. Si es un índice compuesto, se utiliza una pseudo numeración hexadecimal, pero comenzando desde el “valor” *G0* y progresando en forma secuencial. Desde el punto de vista del usuario, los índices secundarios tienen un nombre asociado. El índice primario, en cambio, no tiene nombre.

Se permite el uso directo de índices con claves compuestas, formadas por varios campos. Otras opciones posibles de un índice Paradox son ignorar las mayúsculas y minúsculas en los campos de cadenas (la opción por omisión), la restricción de claves únicas y la posibilidad de ordenación en sentido descendente.

Paradox permite la definición de relaciones de integridad referencial. La siguiente imagen muestra el diálogo de Database Desktop que lo hace posible:



El motor de datos crea automáticamente un índice secundario sobre las columnas de la tabla dependiente que participan en una restricción de integridad. Si la restricción está basada en una sola columna, el índice recibe el nombre de la misma. La tabla que incluyo a continuación describe la implementación en Paradox del comportamiento de la integridad referencial respecto a actualizaciones en la tabla maestra:

	Borrados	Modificaciones
Prohibir la operación	Sí	Sí
Propagar en cascada	No	Sí
Asignar valor por omisión	No	-

No se permiten los borrados en cascada. No obstante, este no es un impedimento significativo, pues dichos borrados pueden implementarse fácilmente en las aplicaciones clientes. En cambio, trate el lector de implementar una propagación en cascada de un cambio de clave cuando existe una integridad referencial...

No fue hasta la aparición de la versión 3.0 del BDE, que acompañó a Delphi 2, que Paradox y dBase contaron con algún tipo de soporte para transacciones. No obstante, este soporte es actualmente muy elemental. Utiliza un *undo log*, es decir, un fichero en el que se van grabando las operaciones que deben deshacerse. Si se desconecta la máquina durante una de estas transacciones, los cambios aplicados a medias quedan grabados, y actualmente no hay forma de deshacerlos. La independencia entre transacciones lanzadas por diferentes procesos es la mínima posible, pues un

proceso puede ver los cambios efectuados por otro proceso aunque éste no haya confirmado aún toda la transacción.

Finalmente, las transacciones locales tienen una limitación importante. La contención entre procesos está implementada mediante bloqueos. Actualmente Paradox permite hasta 255 bloqueos por tabla, y dBase solamente 100. De modo que una transacción en Paradox puede modificar directamente un máximo de 255 registros por tabla.

El formato DBF7

¿Quién no ha tenido que trabajar, en algún momento y de una forma u otra, con los conocidos ficheros DBF? La popularidad de este formato se desarrolló en paralelo con la aparición de los PCs y la propagación de MS-DOS. En el principio, era propiedad de una compañía llamada Ashton-Tate. Ashton fue el empresario que compró el software a su desarrollador, y tenía un loro llamado Tate, ¿o era al revés? Después a esta empresa le fueron mal las cosas y Borland adquirió sus productos. Aunque muchos interpretaron esta acción como un reconocimiento a los “méritos” técnicos de dBase, se trataba en realidad de adquirir otro producto de bases de datos, que en aquel momento era un perfecto desconocido: InterBase. Hay que advertir que InterBase había sido a su vez comprado por Ashton-Tate a sus desarrolladores originales, una efímera compañía de nombre Groton Database Systems, cuyas siglas aún perduran en la extensión por omisión de los ficheros de InterBase: *gdb*.

El formato DBF es muy sencillo ... y muy malo. Fue diseñado como “la base de datos de los pobres”. Cada tabla, al igual que sucede en Paradox, se representa en un conjunto de ficheros. El fichero de extensión *dbf* contiene a la vez la descripción de los campos y los registros, estos últimos de longitud fija. Los registros se almacenan de forma secuencial, sin importar las fronteras de bloques y la pérdida de eficiencia que esto causa. Los tipos de datos originales se representaban en formato ASCII legible, en vez de utilizar su codificación binaria, más compacta. De este modo, las fechas se representaban en ocho caracteres, con el formato *AAAAMMDD*, y un valor entero de 16 bits ocupaba 40 bits, o 48 si se representaba el signo. Por otra parte, hasta fechas recientes el único tipo de cadena implementado tenía longitud fija, obligando a los programadores a usar exhaustivamente la función *Trim*.

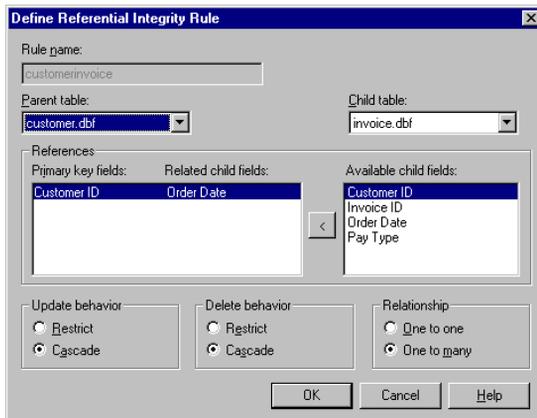
Se pueden utilizar campos de longitud variable, que en principio estuvieron limitados al tipo texto, o *memo*. Los valores de estos campos se almacenan en un fichero paralelo de extensión *dbt*. En este fichero los datos sí respetan la frontera de bloque. Por omisión, el tamaño de estos bloques es de 1024 bytes, pero este parámetro es configurable. Los índices, por su parte, se almacenan todos en un fichero de extensión *mdx*. Aunque, en principio, una tabla puede tener varios ficheros *mdx* asociados, solamente los índices que se encuentran en el fichero que tiene el mismo nombre que la

tabla se actualizan automáticamente, por lo que en la práctica solamente cuenta este fichero, denominado *índice de producción*.

Una de las peculiaridades de dBase es la forma en que se implementan los índices sobre varias columnas. dBase no tiene índices compuestos, al estilo de Paradox y de los sistemas SQL. Como alternativa ofrece los índices basados en expresiones: si queremos un índice compuesto por las columnas *Nombre* y *Apellidos*, tenemos que definir un índice por la expresión *Nombre+Apellidos*. Puede parecer que ésta es una opción muy potente, pero en la práctica el uso de los índices sobre expresiones se reduce a sustituir los índices compuestos de otros sistemas. Además, veremos que trabajar con estos índices en C++ Builder es bastante penoso. ¿Claves primarias, integridad referencial? Nada de eso tenía originalmente el formato DBF. Cuesta más trabajo rediseñar una aplicación escrita en C++ Builder con dBase para que se ejecute en un sistema cliente/servidor que si hubiera sido desarrollada para Paradox.

¿Algo a favor de este formato? Las dos cualidades que mencionaré son consecuencia de la simplicidad de sus ficheros. En primer lugar, las operaciones de actualización son un poco más rápidas en dBase que en Paradox. Y, lo más importante, al existir “menos cosas para romperse”, hay más estabilidad en este formato y más tolerancia respecto a fallos en el sistema operativo.

Con la aparición de Visual dBase 7 junto a la versión 4.50 del BDE, el formato DBF fue renovado en gran medida. Una de las novedades más importantes es la incorporación de relaciones de integridad referencial, que incluyen los borrados en cascada, como se muestra en el siguiente cuadro de diálogo del Administrador de Bases de Datos de Visual dBase 7:



También se han añadido tipos de datos representados en forma binaria, condiciones de verificación, que consisten en expresiones lógicas que deben satisfacer siempre las filas de una tabla, valores por omisión, etc. Lamentablemente, seguimos con los índi-

ces de expresiones como sustitutos de los índices compuestos, y el límite de registros modificados por transacción sigue siendo 100.

Crterios para evaluar un servidor SQL

¿Cómo saber cuál es el sistema de bases de datos cliente/servidor más adecuado para nuestros objetivos? En esta sección trato de establecer criterios de comparación para estos sistemas. No hay dos aplicaciones con las mismas necesidades, así que el hecho de que un sistema no ofrezca determinada opción no lo invalida como la solución que usted necesita. Recuerde que no siempre más es mejor.

- **Plataformas soportadas**

¿En qué sistema operativo debe ejecutarse el servidor? La respuesta es importante por dos razones. La primera: nos da una medida de la flexibilidad que tenemos a la hora de seleccionar una configuración de la red. Si vamos a instalar una aplicación en una empresa que tiene toda su red basada en determinado servidor con determinado protocolo, no es recomendable que abandonen todo lo que tienen hecho para que el recién llegado pueda ejecutarse.

En segundo lugar, no todos los sistemas operativos se comportan igual de eficientes actuando como servidores de bases de datos. Esto tiene que ver sobre todo con la implementación que realiza el SO de la concurrencia. Mi favorito en este sentido es UNIX: un InterBase, un Oracle, un Informix, ejecutándose sobre cualquier “sabor” de UNIX (HP-UX, AIX, Solaris).

Puede suceder, por supuesto, que el mantenimiento de la red no esté en manos de un profesional dedicado a esta área. En tal caso, hay que reconocer que es mucho más sencillo administrar un Windows NT.

- **Soporte de tipos de datos y restricciones**

En realidad, casi todos los sistemas SQL actuales tienen tipos de datos que incluyen a los especificados en el estándar de SQL, excepto determinados casos patológicos. De lo que se trata sobre todo es la implementación de los mismos. Por ejemplo, no todas los formatos de bases de datos implementan con la misma eficiencia el tipo *VARCHAR*, que almacena cadenas de longitud variable. La longitud máxima es uno de los parámetros que varían, y el formato de representación: si siempre se asigna un tamaño fijo (el máximo) para estos campos, o si la longitud total del registro puede variar.

Más importante es el soporte para validaciones declarativas, esto es, verificaciones para las cuales no hay que desarrollar código especial. Dentro de este apartado entran las claves primarias, claves alternativas, relaciones de integridad referencial, dominios, chequeos a nivel de fila, etc. Un elemento a tener en cuenta, por ejemplo, es si se permite o no la propagación en cascada de modificaciones en la tabla maestra de una relación de integridad referencial. En caso positivo,

esto puede ahorrarnos bastante código en *triggers* y permitirá mejores modelos de bases de datos.

- **Lenguaje de triggers y procedimientos almacenados**

Este es uno de los criterios a los que concedo mayor importancia. El éxito de una aplicación, o un conjunto de aplicaciones, en un entorno C/S depende en gran medida de la forma en que dividamos la carga de la aplicación entre el servidor de datos y los clientes. En este punto es donde podemos encontrar mayores diferencias entre los sistemas SQL, pues no hay dos dialectos de este lenguaje que sean exactamente iguales.

Debemos fijarnos en si el lenguaje permite *triggers* a nivel de fila o de operación, o de ambos niveles. Un *trigger* a nivel de fila se dispara antes o después de modificar una fila individual. Por el contrario, un *trigger* a nivel de operación se dispara después de que se ejecute una operación completa que puede afectar a varias filas. Recuerde que SQL permite instrucciones como la siguiente:

```
delete from Clientes
where Ciudad in ("Pompeya", "Herculano")
```

Si la base de datos en cuestión sólo ejecuta sus *triggers* al terminar las operaciones, será mucho más complicada la programación de los mismos, pues más trabajo costará restablecer los valores anteriores y posteriores de cada fila particular. También debe tenerse en cuenta las posibilidades de extensión del lenguaje, por ejemplo, incorporando funciones definidas en otros lenguajes, o el poder utilizar servidores de automatización COM o CORBA, si se permiten o no llamadas recursivas, etc.

- **Implementación de transacciones: recuperación y aislamiento**

Este es mi otro criterio de análisis preferido. Las transacciones ofrecen a las bases de datos la consistencia necesaria para que operaciones parciales no priven de sentido semántico a los datos almacenados. Al constituir las unidades básicas de procesamiento del servidor, el mecanismo que las soporta debe encargarse también de aislar a los usuarios entre sí, de modo que la secuencia exacta de pasos concurrentes que realicen no influya en el resultado final de sus acciones. Por lo tanto, hay dos puntos en los que centrar la atención: cómo se implementa la atomicidad (la forma en que el sistema deshace operaciones inconclusas), y el método empleado para aislar entre sí las transacciones concurrentes.

- **Segmentación**

Es conveniente poder distribuir los datos de un servidor en distintos dispositivos físicos. Al situar tablas en distintos discos, o segmentos de las propias tablas, podemos aprovechar la concurrencia inherente a la existencia de varios controladores físicos de estos medios de almacenamiento. Esta opción permite, además,

superar las restricciones impuestas por el tamaño de los discos en el tamaño de la base de datos.

- **Replicación**

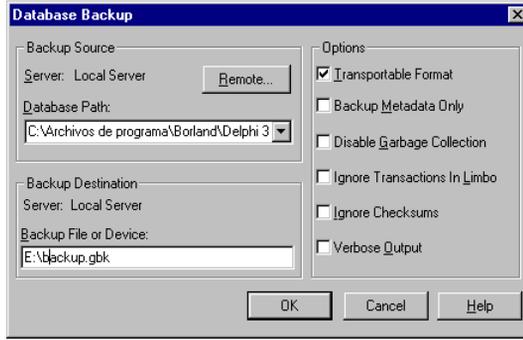
Uno de los usos principales de la replicación consiste en aislar las aplicaciones que realizan mantenimientos (transacciones OLTP) de las aplicaciones para toma de decisiones (transacciones DSS). Las aplicaciones con fuerte base OLTP tienden a modificar en cada transacción un pequeño número de registros. Las aplicaciones DSS se caracterizan por leer grandes cantidades de información, siendo poco probable que escriban en la base de datos. En sistemas de bases de datos que implementan el aislamiento de transacciones mediante bloqueos, ambos tipos de transacciones tienen una coexistencia difícil. Por lo tanto, es conveniente disponer de réplicas de la base de datos sólo para lectura, y que sea a estos clones a quienes se refieran las aplicaciones DSS.

Me da un poco de vergüenza añadir el último factor a la lista anterior: el precio. Todos queremos lo mejor, pero no siempre estamos dispuestos a pagar por eso. Así que muchas veces una decisión de compra representa un balance entre la calidad y el precio ... como en todo.

InterBase

Los antiguos griegos representaban a la Fama y a la Fortuna como caprichosas diosas que otorgaban sus favores muchas veces a quienes menos lo merecían; estos griegos de antes, la verdad, eran bastante misóginos, entre otras cosas. En cualquier caso, InterBase debe haber tenido algún altercado con la señorita Fama, porque no tiene la popularidad que merecería por sus muchas virtudes.

Una de estas virtudes es que se puede instalar el servidor en muchos sistemas operativos: Windows NT y 9x, NetWare, Solaris, HP-UX, SCO, Linux... Otra es que en cualquiera de estos sistemas ocupa muy poco espacio, del orden de los 10 ó 20 MB. El mantenimiento de un servidor, una vez instalado, es también mínimo. Cada base de datos se sitúa en uno o más ficheros, casi siempre de extensión *gdb*. Estos ficheros crecen automáticamente cuando hace falta más espacio, y no hay que preocuparse por reservar espacio adicional para registrar los cambios efectuados por las transacciones, como en otros sistemas. Se pueden realizar copias de seguridad de una base de datos “en caliente”, mientras que otros sistemas requieren que no existan usuarios activos para efectuar esta operación. Y la recuperación después de un fallo de hardware es sencilla e inmediata: vuelva a encender el servidor.



Tal sencillez tiene un precio, y es que actualmente InterBase no implementa directamente ciertas opciones avanzadas de administración, como la segmentación y la replicación. Esta última debe ser implementada manualmente, por medio de *triggers* definidos por el diseñador de la base de datos, y con la ayuda de un proceso en segundo plano que vaya grabando los datos del original a la réplica. No obstante, InterBase permite definir directamente copias en espejo (*mirrors*) de una base de datos, de forma tal que existan dos copias sincronizadas de una misma base de datos en discos duros diferentes del mismo servidor. De este modo, si se produce un fallo de hardware en uno de los discos o controladores, la explotación de la base de datos puede continuar con la segunda copia.

En lo que atañe a los tipos de datos, InterBase implementa todos los tipos del estándar SQL con una excepción. Este lenguaje define tres tipos de campos para la fecha y la hora: *DATE*, para fechas, *TIME*, para horas, y *TIMESTAMP* para almacenar conjuntamente fechas y horas. InterBase solamente tiene *DATE*, pero como equivalente al *TIMESTAMP* del estándar. Esto en sí no conlleva problemas, a no ser que haya que escribir una aplicación que acceda indistintamente a bases de datos de InterBase y de cualquier otro sistema SQL. Pero cuando estudiemos el uso del Diccionario de Datos de C++ Builder, veremos cómo resolver esta nimiedad.

InterBase tiene, hoy por hoy, una de las implementaciones más completas de las restricciones de integridad referencial, pues permite especificar declarativamente la propagación en cascada de borrados y modificaciones:

	Borrados	Modificaciones
Prohibir la operación	Sí	Sí
Propagar en cascada	Sí	Sí
Asignar valor por omisión	Sí	-

Por supuesto, tenemos todas las restricciones de unicidad, claves primarias, validaciones a nivel de registro (cláusulas **check**). Estas últimas son más potentes que en el resto de los sistemas, pues permiten realizar comprobaciones que involucren a registros de otras tablas.

Los *triggers* de InterBase se ejecutan antes o después de cada operación de actualización, fila por fila, que es como Dios manda. Por otra parte, tiene un lenguaje de procedimientos almacenados muy completo, que permite llamadas recursivas, y la definición de procedimientos de selección, que devuelven más de un registro de datos por demanda. Todo ello se integra con un mecanismo de excepciones muy elegante, que compagina muy bien con las técnicas transaccionales. Es posible, además, extender el conjunto de funciones del lenguaje mediante módulos dinámicos, DLLs en el caso de las versiones para Windows y Windows NT.

Pero la característica más destacada de InterBase es la forma en la que logra implementar el acceso concurrente a sus bases de datos garantizando que, en lo posible, cada usuario no se vea afectado por las acciones del resto. Casi todos los sistemas existentes utilizan, de una forma u otra, bloqueos para este fin. InterBase utiliza la denominada *arquitectura multigeneracional*, en la cual cada vez que una transacción modifica un registro se genera una nueva versión del mismo. Teóricamente, esta técnica permite la mayor cantidad de acciones concurrentes sobre las bases de datos. En la práctica, la implementación de InterBase es muy eficiente y confiable. Todo esto lo estudiaremos en el capítulo 12, que trata acerca de las transacciones.

La última versión de InterBase, en el momento en que escribo estas líneas, es la 5.5, y acompaña a C++ Builder 4: en la versión Profesional con licencias para InterBase Local, y en la Cliente/Servidor, con licencias para un servidor sobre Windows 9x y NT.

Microsoft SQL Server

Toda semejanza en la explicación de las características de Microsoft SQL Server con las de Sybase será algo más que una simple coincidencia. Realmente MS SQL Server comenzó como un derivado del servidor de Sybase, por lo que la arquitectura de ambos es muy parecida. De modo que gran parte de lo que se diga en esta sección sobre un sistema, vale para el otro.

Es muy fácil tropezarse por ahí con MS SQL Server. Hay incluso quienes lo tienen instalado y aún no se han dado cuenta. Microsoft tiene una política de distribución bastante agresiva para este producto, pues lo incluye en el paquete BackOffice, junto a su sistema operativo Windows NT y unos cuantos programas más. Como puede imaginar el lector, SQL Server 6.5 puede ejecutarse en Windows NT y en Windows NT. Por fortuna, la versión 7.0 ofrece un servidor para Windows 9x ... aunque obliga al usuario a instalar primeramente Internet Explorer 4.

Lo primero que llama la atención es la cantidad de recursos del sistema que consume una instalación de SQL Server. La versión 6.5 ocupa de 70 a 90MB, mientras que la

versión 7 llega a los 180MB de espacio en disco. ¿La explicación? Asistentes, y más asistentes: en esto contrasta con la austeridad espartana de InterBase.

A pesar de que la instalación del servidor es relativamente sencilla, su mantenimiento es bastante complicado, sobre todo en la versión 6.5. Cada base de datos reside en uno o más *dispositivos físicos*, que el fondo no son más que vulgares ficheros. Estos dispositivos ayudan a implementar la segmentación, pero no crecen automáticamente, por lo que el administrador del sistema deberá estar pendiente del momento en que los datos están a punto de producir un desbordamiento. A diferencia de InterBase, para cada base de datos hay que definir explícitamente un *log*, o registro de transacciones, que compite en espacio con los datos verdaderos, aunque este registro puede residir en otro dispositivo (lo cual se recomienda).

Aunque los mecanismos de *logging*, en combinación con los bloqueos, son los más frecuentes en las bases de datos relacionales como forma de implementar transacciones atómicas, presentan claras desventajas en comparación con la arquitectura multigeneracional. En primer lugar, no se pueden realizar copias de seguridad con usuarios conectados a una base de datos. Los procesos que escriben bloquean a los procesos que se limitan a leer información, y viceversa. Si se desconecta físicamente el servidor, es muy probable que haya que examinar el registro de transacciones antes de volver a echar a andar las bases de datos. Por último, hay que estar pendientes del crecimiento de estos ficheros. Hay un experimento muy sencillo que realizo con frecuencia en InterBase, poblar una tabla con medio millón de registros, que nunca he logrado repetir en SQL Server, por mucho que he modificado parámetros de configuración.

Hay que reconocer que esta situación mejora un poco con la versión 7, pues desaparece el concepto de dispositivo, siendo sustituido por el de fichero del propio sistema operativo: las bases de datos se sitúan en ficheros de extensión *mdf* y *ndf*, los registros de transacciones, en ficheros *ldf*. Este nuevo formato permite que los ficheros de datos y de transacciones crezcan dinámicamente, por demanda.

Otro grave problema de versiones anteriores que soluciona la nueva versión es la granularidad de los bloqueos. Antes, cada modificación de un registro imponía un bloqueo a toda la página en que éste se encontraba. Además, las páginas tenían un tamaño fijo de 2048 bytes, lo que limitaba a su vez el tamaño máximo de un registro. En la versión 6.5 se introdujo el bloqueo a nivel de registro ... pero únicamente para las inserciones, que es cuando menos hacen falta. Finalmente, la versión 7 permite siempre bloqueos de registros, que pueden escalar por demanda a bloqueos de página o a nivel de tablas, y aumenta el tamaño de página a 8192 bytes. No obstante, este tamaño sigue sin poder ajustarse.

Microsoft SQL Server ofrece extrañas extensiones a SQL que solamente sirven para complicarnos la vida a usted y mí. Por ejemplo, aunque el SQL estándar dice que por

omisión una columna admite valores nulos, esto depende en SQL Server del estado de un parámetro, ¡que por omisión produce el efecto contrario! La implementación de la integridad referencial en este sistema es bastante pobre, pues solamente permite restringir las actualizaciones y borrados en la tabla maestra; nada de propagación en cascada y otras alegrías. También es curioso que SQL Server no crea automáticamente índices secundarios sobre las tablas que contienen claves externas.

	Borrados	Modificaciones
Prohibir la operación	Sí	Sí
Propagar en cascada	No	No
Asignar valor por omisión	No	-

Otro de los aspectos negativos de SQL Server es su lenguaje de *triggers* y procedimientos almacenados, llamado Transact-SQL, que es bastante excéntrico respecto al resto de los lenguajes existentes y a la propuesta de estándar. Uno puede acostumbrarse a soberanas tonterías tales como obligar a que todas las variables locales y parámetros comiencen con el carácter @. Pero es bastante difícil programar determinadas reglas de empresa cuando los *triggers* se disparan solamente después de instrucciones completas.

En versiones anteriores del BDE, el nivel superior de aislamiento de transacciones solamente se alcanzaba en bases de datos abiertas en modo sólo lectura. Actualmente es posible activar este nivel desde C++ Builder sin problemas, aunque en el capítulo dedicado a este sistema de bases de datos estudiaremos trucos para garantizar lecturas repetibles sin necesidad de afectar demasiado a otros usuarios.

Oracle

Oracle es uno de los abuelos en el negocio de las bases de datos relacionales; el otro es DB2, de IBM⁴. Este sistema es otra de las apuestas seguras en el caso de tener que elegir un servidor de bases de datos. ¿Su principal desventaja? Resulta que no es de carácter técnico, sino que tiene que ver con una política de precios altos, alto coste de la formación y del mantenimiento posterior del sistema. Pero si usted puede permitirse el lujo...

Piense en una plataforma de hardware ... ¿ya?, pues Oracle funciona en la misma. Los ejemplos para Oracle de este libro han sido desarrollados, concretamente, con Personal Oracle, versiones 7.3 y 8.0, para Windows 95. Este es un servidor muy estable, quizás algo lento en establecer la conexión a la base de datos, que a veces cuesta un poco instalar adecuadamente (sobre todo por las complicaciones típicas de TCP/IP),

⁴ Debe haber una explicación (nunca me la han contado) a esta pulsión freudiana del gigante azul para que sus productos siempre sean “segundos”: DB2, OS/2, PS/2 ...

pero una vez en funcionamiento va de maravillas. Así que con Oracle no tiene pre-textos para no llevarse trabajo a casa.

Oracle permite todas las funciones avanzadas de un servidor SQL serio: segmentación, replicación, etc. Incluso puede pensarse que tiene demasiados parámetros de configuración. La parte principal del control de transacciones se implementa mediante bloqueos y registros de transacciones, aunque el nivel de aislamiento superior se logra mediante copias sólo lectura de los datos. Por supuesto, el nivel mínimo de granularidad de estos bloqueos es a nivel de registro.

¿Tipos de datos? Todos los que usted desee. ¿Restricciones **check**? No tan generales como las de InterBase, pero quedan compensadas por la mayor abundancia de funciones predefinidas. Hasta la versión 7.3, Oracle implementaba solamente la propagación en cascada de borrados para la integridad referencial, como muestra la siguiente tabla:

	Borrados	Modificaciones
Prohibir la operación	Sí	Sí
Propagar en cascada	Sí	No
Asignar valor por omisión	Sí	-

Las extensiones procedimentales a SQL, denominadas PL/SQL, conforman un lenguaje potente, que permite programar *paquetes (packages)* para la implementación de tipos de datos abstractos. Con la versión 8, incluso, se pueden definir tipos de clases, u *objetos*. Esta última extensión no es, sin embargo, lo suficientemente general como para clasificar a este sistema como una base de datos orientada a objetos, en el sentido moderno de esta denominación. Uno de los puntos fuertes de la versión 4.0 de C++ Builder es la posibilidad de trabajar con las extensiones de objetos de Oracle 8.

Como pudiera esperarse, el lenguaje de *triggers* es muy completo, y permite especificarlos tanto a nivel de fila como de operación. Hay montones de funciones utilizables desde SQL, y curiosas extensiones al lenguaje consulta, como la posibilidad de realizar determinados tipos de clausuras transitivas.

Otros sistemas de uso frecuente

Evidentemente, es imposible hablar con autoridad acerca de todos los formatos de bases de datos existentes en el mercado, y en las secciones anteriores me he limitado a presentar aquellos sistemas con los que he trabajado con mayor frecuencia. Sin embargo, gracias a las particularidades de mi actual ocupación, he podido ver en funcionamiento a muchos de los restantes sistemas SQL con los que C++ Builder nos permite trabajar directamente.

Por ejemplo, DB2, de IBM. Antes mencioné que este sistema y Oracle eran los dos sistemas que más tiempo llevaban en este negocio, y los frutos de esta experiencia se dejan notar también en DB2. Existen actualmente versiones de DB2 para una amplia gama de sistemas operativos. El autor lo ha visto funcionar sobre OS/2, Windows NT y Windows 95, teniendo una versión de evaluación sobre este último sistema. Por supuesto, estas no son las únicas plataformas sobre las que puede ejecutarse.

La arquitectura de DB2 es similar a la de Oracle, a la que se parece la de MS SQL Server, que es similar a la de Sybase SQL Server.. En realidad, la concepción de estos sistemas está basada en un proyecto experimental de IBM, denominado System-R, que fue la primera implementación de un sistema relacional. En este proyecto se desarrollaron o perfeccionaron técnicas como los identificadores de registros, los mecanismos de bloqueos actuales, registros de transacciones, índices basados en árboles balanceados, los algoritmos de optimización de consultas, etc. Así que también podrá usted esperar de DB2 la posibilidad de dividir en segmentos sus bases de datos, de poder realizar réplicas y de disponer de transacciones atómicas y coherentes. El mantenimiento de las bases de datos de DB2 puede ser todo lo simple que usted desee (sacrificando algo el rendimiento), o todo lo complicado que le parezca (a costa de su cuero cabelludo). El lenguaje de *triggers* y procedimientos almacenados es muy completo, y similar al de Oracle e InterBase, como era de esperar. La única pega que le puedo poner a DB2 es que la instalación de clientes es bastante pesada, y para poder conectar una estación de trabajo hay que realizar manualmente un proceso conocido como *catalogación*. Pero esto mismo le sucede a Oracle con su SQL Net.

Otro sistema importante es Informix, que está bastante ligado al mundo de UNIX, aunque en estos momentos existen versiones del servidor para Windows NT. Su arquitectura es similar a la de los sistemas antes mencionados.

Finalmente, quiero referirme aunque sea de pasada a otras bases de datos “no BDE”. Tenemos, por ejemplo, la posibilidad de trabajar con bases de datos de AS/400. Aunque el motor de datos que viene con C++ Builder no permite el acceso directo a las mismas, podemos programar para estas bases de datos colocando una pasarela DB2 como interfaz. No obstante, el producto C++ Builder/400 sí que nos deja saltarnos las capas intermedias, logrando mayor eficiencia a expensas de la pérdida de portabilidad. También está muy difundido Btrieve, una base de datos que inició su vida como un sistema navegacional, pero que en sus últimas versiones ha desarrollado el producto Pervasive SQL, que es un motor de datos cliente/servidor relacional. Lamentablemente, tampoco está soportado directamente por el motor de datos de C++ Builder, por el momento.

Breve introducción a SQL

CUANDO IBM DESARROLLÓ EL PRIMER PROTOTIPO DE base de datos relacional, el famoso System R, creó en paralelo un lenguaje de definición y manipulación de datos, llamado QUEL. La versión mejorada de este lenguaje que apareció un poco más tarde se denominó, un poco en broma, SEQUEL. Finalmente, las siglas se quedaron en SQL: *Structured Query Language*, o Lenguaje de Consultas Estructurado. Hay quien sigue pronunciando estas siglas en inglés como *sequel*, es decir, secuela.

La estructura de SQL

Las instrucciones de SQL se pueden agrupar en dos grandes categorías: las instrucciones que trabajan con la estructura de los datos y las instrucciones que trabajan con los datos en sí. Al primer grupo de instrucciones se le denomina también el *Lenguaje de Definición de Datos*, en inglés *Data Definition Language*, con las siglas DDL. Al segundo grupo se le denomina el *Lenguaje de Manipulación de Datos*, en inglés *Data Manipulation Language*, o DML. A veces las instrucciones que modifican el acceso de los usuarios a los objetos de la base de datos, y que en este libro se incluyen en el DDL, se consideran pertenecientes a un tercer conjunto: el *Lenguaje de Control de Datos*, *Data Control Language*, ó DCL.

En estos momentos existen estándares aceptables para estos tres componentes del lenguaje. El primer estándar, realizado por la institución norteamericana ANSI y luego adoptada por la internacional ISO, se terminó de elaborar en 1987. El segundo, que es el que está actualmente en vigor, es del año 1992. En estos momentos está a punto de ser aprobado un tercer estándar, que ya es conocido como SQL-3.

Las condiciones en que se elaboró el estándar de 1987 fueron especiales, pues el mercado de las bases de datos relacionales estaba dominado por unas cuantas compañías, que trataban de imponer sus respectivos dialectos del lenguaje. El acuerdo final dejó solamente las construcciones que eran comunes a todas las implementaciones; de este modo, nadie estaba obligado a reescribir su sistema para no quedarse sin la certificación. También se definieron diferentes niveles de conformidad para

facilitarles las cosas a los fabricantes; si en algún momento alguien le intenta vender un sistema SQL alabando su conformidad con el estándar, y descubre en letra pequeña la aclaración “compatible con el nivel de entrada (*entry-level*)”, tenga por seguro que lo están camelando. Este estándar dejó fuera cosas tan necesarias como las definiciones de integridad referencial. Sin embargo, introdujo el denominado *lenguaje de módulos*, una especie de interfaz para desarrollar funciones en SQL que pudieran utilizarse en programas escritos en otros lenguajes.

El estándar del 92 se ocupó de la mayoría de las áreas que quedaron por cubrir en el 87. Sin embargo, no se hizo nada respecto a recursos tales como los procedimientos almacenados, los *triggers* o disparadores, y las excepciones, que permiten la especificación de reglas para mantener la integridad y consistencia desde un enfoque *imperativo*, en contraste con el enfoque *declarativo* utilizado por el DDL, el DCL y el DML. Dedicaremos un capítulo al estudio de estas construcciones del lenguaje. En este preciso momento, cada fabricante tiene su propio dialecto para definir procedimientos almacenados y disparadores. El estándar conocido como SQL-3 se encarga precisamente de unificar el uso de estas construcciones del lenguaje.

En este capítulo solamente nos ocuparemos de los lenguajes de definición y control de datos. El lenguaje de manipulación de datos se trata en el capítulo siguiente. Más adelante nos ocuparemos del lenguaje de *triggers* y procedimientos almacenados.

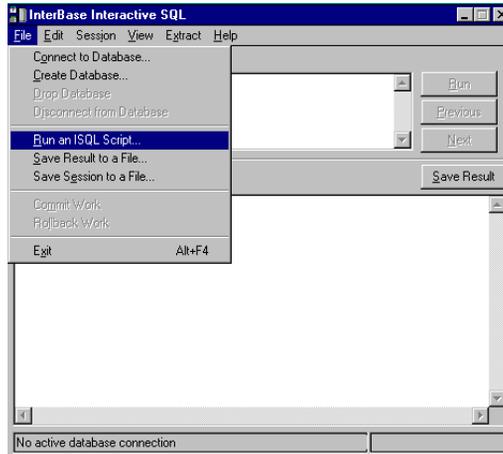
Para seguir los ejemplos de este libro...

Para poder mostrar todas las posibilidades del lenguaje SQL necesitamos un sistema de bases de datos potente, que admita la mayor parte posible de las construcciones sintácticas definidas por el lenguaje. Aunque Paradox y dBase pueden ser abordados mediante SQL, el intérprete ofrecido por el BDE tiene limitaciones, sobre todo en el lenguaje de definición de datos. Por lo tanto, necesitamos algún sistema cliente/servidor para seguir los ejemplos. ¿Cuál de los muchos disponibles? Para este capítulo y para el siguiente, casi da lo mismo el sistema elegido, pues las implementaciones de los sublenguajes DDL y DML son bastante similares en casi todos los dialectos existentes de SQL. Donde realmente necesitaremos aclarar con qué dialecto estaremos tratando en cada momento, será en el capítulo sobre *triggers* y procedimientos almacenados.

Así que si usted tiene un Oracle a mano, utilice Oracle; si tiene Informix, utilícelo. ¿Pero qué pasa si nunca ha trabajado con uno de esos sistemas? Pues que siempre tendremos a mano a InterBase, el sistema de bases de datos SQL de Borland. Si el lector tiene instalada una de las versiones cliente/servidor o profesional de C++ Builder 1 (la versión Desktop no vale), tendrá también instalado el servidor local de InterBase. Si está utilizando C++ Builder 3 ó 4, debe realizar la instalación por separado. En cualquier caso, tomaré como punto de partida a InterBase para los primeros

tres capítulos. En estos capítulos iniciales, presentaré las características fundamentales del lenguaje SQL: el lenguaje de definición de datos en el primer capítulo, en el siguiente, el lenguaje de consultas y manipulación de datos, y para finalizar, las extensiones procedimentales para *triggers* y procedimientos almacenados. Luego dedicaré un par de capítulos a las particularidades de Oracle, MS SQL Server y DB2.

InterBase viene acompañado por la aplicación Windows ISQL. Con esta utilidad podemos crear y borrar bases de datos de InterBase, conectarnos a bases de datos existentes y ejecutar todo tipo de instrucciones SQL sobre ellas. Las instrucciones del lenguaje de manipulación de datos, y algunas del lenguaje de definición, pueden ejecutarse directamente tecleando en un cuadro de edición multilíneas y pulsando un botón. El resultado de la ejecución aparece en un control de visualización situado en la parte inferior de la ventana.



Para las instrucciones DDL más complejas y la gestión de procedimientos, disparadores, dominios y excepciones es preferible utilizar *scripts* SQL. Un *script* es un fichero de texto, por lo general de extensión *sql*, que contiene una lista de instrucciones arbitrarias de este lenguaje separadas entre sí por puntos y comas. Este fichero debe ejecutarse mediante el comando de menú *File | Run an ISQL Script*. Las instrucciones SQL se van ejecutando secuencialmente, según el orden en que se han escrito. Por omisión, los resultados de la ejecución del *script* también aparecen en el control de visualización de la ventana de Windows ISQL.

Por supuesto, también puede utilizarse la utilidad SQL Explorer del propio C++ Builder para ejecutar instrucciones individuales sobre cualquier base de datos a la que deseemos conectarnos. También podemos utilizar Database Desktop si queremos realizar pruebas con Paradox y dBase.

La creación y conexión a la base de datos

Estamos en InterBase, utilizando Windows ISQL. ¿Cómo nos conectamos a una base de datos de InterBase para comenzar a trabajar? Basta con activar el comando de menú *File | Connect to database*. Si ha instalado InterBase en un servidor remoto y tiene los SQL Links que vienen con C++ Builder cliente/servidor, puede elegir la posibilidad de conectarse a ese servidor remoto. En cualquier caso, puede elegir el servidor local. Cuando especificamos un servidor remoto, tenemos que teclear el nombre completo del fichero de base de datos en el servidor; no necesitamos tener acceso al fichero desde el sistema operativo, pues es el servidor de InterBase el que nos garantiza el acceso al mismo. Si estamos utilizando el servidor local, las cosas son más fáciles, pues contamos con un botón *Browse* para explorar el disco. Las bases de datos de InterBase se sitúan por lo general en un único fichero de extensión *gdb*; existe, no obstante, la posibilidad de distribuir información en ficheros secundarios, lo cual puede ser útil en servidores con varios discos duros. Los otros datos que tenemos que suministrar a la conexión son el nombre de usuario y la contraseña. El nombre de usuario inicial en InterBase es, por omisión, *SYSDBA*, y su contraseña es *masterkey*. Respete las mayúsculas y minúsculas, por favor. Una vez que acepte el cuadro de diálogo, se intentará la conexión. En cualquier momento de la sesión podemos saber a qué base de datos estamos conectados mirando la barra de estado de la ventana.

El mismo mecanismo puede utilizarse para crear una base de datos interactivamente. La diferencia está en que debemos utilizar el comando *File | Create database*. Sin embargo, necesitamos saber también cómo podemos crear una base de datos y establecer una conexión utilizando instrucciones SQL. La razón es que todo *script* SQL debe comenzar con una instrucción de creación de bases de datos o de conexión. La más sencilla de estas instrucciones es la de conexión:

```
connect "C:\Marteens\IntrBase\Examples\Prueba.GDB"  
user "SYSDBA" password "masterkey";
```

Por supuesto, el fichero mencionado en la instrucción debe existir, y el nombre de usuario y contraseña deben ser válidos. Observe el punto y coma al final, para separar esta instrucción de la próxima en el *script*.

La instrucción necesaria para crear una base de datos tiene una sintaxis similar. El siguiente ejemplo muestra el ejemplo más común de creación:

```
create database "C:\Marteens\IntrBase\Examples\Prueba.GDB"  
user "SYSDBA" password "masterkey"  
page_size 2048;
```

InterBase, y casi todos los sistemas SQL, almacenan los registros de las tablas en bloques de longitud fija, conocidos como *páginas*. En la instrucción anterior estamos

redefiniendo el tamaño de las páginas de la base de datos. Por omisión, InterBase utiliza páginas de 1024 bytes. En la mayoría de los casos, es conveniente utilizar un tamaño mayor de página; de este modo, el acceso a disco es más eficiente, entran más claves en las páginas de un índice con lo cual disminuye la profundidad de estos, y mejora también el almacenamiento de campos de longitud variable. Sin embargo, si sus aplicaciones trabajan con pocas filas de la base de datos, como la típica aplicación del cajero automático, puede ser más conveniente mantener un tamaño pequeño de página, pues la lectura de éstas tarda entonces menos tiempo, y el *buffer* puede realizar las operaciones de reemplazo de páginas en memoria más eficientemente.

También podemos indicar el tamaño inicial de la base de datos en páginas. Normalmente esto no hace falta, pues InterBase hace crecer automáticamente el fichero *gdb* cuando es necesario. Pero si tiene en mente insertar grandes cantidades de datos sobre la base recién creada, puede ahorrar el tiempo de crecimiento utilizando la opción **length**. La siguiente instrucción crea una base de datos reservando un tamaño inicial de 1 MB:

```
create database "C:\Marteens\IntrBase\Examples\Prueba.GDB"
user "SYSDBA" password "masterkey"
page_size 2048 length 512
default character set "ISO8859_1";
```

Esta instrucción muestra también cómo especificar el *conjunto de caracteres* utilizado por omisión en la base de datos. Más adelante, se pueden definir conjuntos especiales para cada tabla, de ser necesario. El conjunto de caracteres determina, fundamentalmente, de qué forma se ordenan alfabéticamente los valores alfanuméricos. Los primeros 127 caracteres de todos los conjuntos de datos coinciden; es en los restantes valores donde puede haber diferencias.

Tipos de datos en SQL

Antes de poder crear tablas, tenemos que saber qué tipos de datos podemos emplear. SQL estándar define un conjunto de tipos de datos que todo sistema debe implementar. Ahora bien, la interpretación exacta de estos tipos no está completamente especificada. Cada sistema de bases de datos ofrece, además, sus propios tipos nativos. Estos son los tipos de datos aceptados por InterBase:

Tipo de dato	Tamaño	Observaciones
char(<i>n</i>), varchar(<i>n</i>)	<i>n</i> bytes	Longitud fija; longitud variable
integer, int	32 bits	
smallint	16 bits	
float	4 bytes	Equivale al tipo homónimo de C
double precision	8 bytes	Equivale al <i>double</i> de C
numeric(<i>prec</i> , <i>esc</i>)	<i>prec</i> =1-15, <i>esc</i> <= <i>prec</i>	Variante “exacta” de <i>decimal</i>

Tipo de dato	Tamaño	Observaciones
decimal(<i>prec, esc</i>)	<i>prec</i> =1-15, <i>esc</i> <= <i>prec</i>	
date	64 bits	Almacena la fecha y la hora
blob	No hay límite	

La diferencia fundamental respecto al estándar SQL tiene que ver con el tipo **date**. SQL estándar ofrece los tipos **date**, **time** y **timestamp**, para representar fecha, hora y la combinación de fecha y hora. El tipo **date** de InterBase corresponde al tipo **timestamp** del SQL estándar.

El tipo de dato **blob** (*Binary Large Object* = Objeto Binario Grande) se utiliza para almacenar información de longitud variable, generalmente de gran tamaño. En principio, a InterBase no le preocupa qué formato tiene la información almacenada. Pero para cada tipo **blob** se define un *subtipo*, un valor entero que ofrece una pista acerca del formato del campo. InterBase interpreta el subtipo 0 como el formato por omisión: ningún formato. El subtipo 1 representa texto, como el tipo memo de otros sistemas. Se pueden especificar subtipos definidos por el programador; en este caso, los valores empleados deben ser negativos. La especificación de subtipos se realiza mediante la cláusula **sub_type**:

```
Comentarios blob sub_type 1
```

Una de las peculiaridades de InterBase como gestor de bases de datos es el soporte de *matrices multidimensionales*. Se pueden crear columnas que contengan matrices de tipos simples, con excepción del tipo **blob**, de hasta 16 dimensiones. Sin embargo, C++ Builder no reconoce directamente este tipo de campos, y debemos trabajar con ellos como si fueran campos BLOB.

Representación de datos en InterBase

¿Qué diferencias hay entre los tipos **char** y **varchar**? Cuando una aplicación graba una cadena en una columna de tipo **varchar**, InterBase almacena exactamente la misma cantidad de caracteres que ha especificado la aplicación, independientemente del ancho máximo de la columna. Cuando se recupera el valor más adelante, la cadena obtenida tiene la misma longitud que la original. Ahora bien, si la columna de que hablamos ha sido definida como **char**, en el proceso de grabación se le añaden automáticamente espacios en blanco al final del valor para completar la longitud de la cadena. Cuando se vuelve a leer la columna, la aplicación recibe estos espacios adicionales.

¿Quiere esto decir que ahorraremos espacio en la base de datos utilizando siempre el tipo **varchar**? ¡Conclusión prematura! InterBase utiliza registros de longitud variable para representar las filas de una tabla, con el fin de empaquetar la mayor cantidad

posible de registros en cada página de la base de datos. Como parte de la estrategia de disminución del tamaño, cuando se almacena una columna de tipo **char** se eliminan automáticamente los espacios en blanco que puede contener al final, y estos espacios se restauran cuando alguien recupera el valor de dicha columna. Más aún: para almacenar un **varchar** es necesario añadir a la propia representación de la cadena un valor entero con la longitud de la misma. Como resultado final, una columna de tipo **varchar** consume más espacio que una de tipo **char**!

¿Para qué, entonces, quiero el tipo **varchar**?, se preguntará el lector. Conviene que recuerde que si utiliza el tipo **char** recibirá valores con espacios en blanco adicionales al final de los mismos, y que tendrá que utilizar frecuentemente la función *TrimRight* para eliminarlos. El tipo **varchar** le ahorra este incordio.

También le será útil conocer cómo InterBase representa los tipos **numeric** y **decimal**. El factor decisivo de la representación es el número de dígitos de la precisión. Si es menor que 5, **numeric** y **decimal** pueden almacenarse dentro de un tipo entero de 16 bits, o **smallint**; si es menor que 10, en un **integer** de 32 bits; en caso contrario, se almacenan en columnas de tipo **double precision**.

Creación de tablas

Como fuente de ejemplos para este capítulo, utilizaremos el típico esquema de un sistema de pedidos. Las tablas que utilizaremos serán las siguientes:

Tabla	Propósito
<i>Clientes</i>	Los clientes de nuestra empresa
<i>Empleados</i>	Los empleados que reciben los pedidos
<i>Articulos</i>	Las cosas que intentamos vender
<i>Pedidos</i>	Pedidos realizados
<i>Detalles</i>	Una fila por cada artículo vendido

Un ejemplo similar lo encontramos en la base de datos de demostración que trae C++ Builder, pero en formato Paradox.

La instrucción de creación de tablas tiene la siguiente sintaxis en InterBase:

```
create table NombreDeTabla [external file NombreFichero] (
    DefColumna [, DefColumna | Restriccion ... ]
);
```

La opción **external file** es propia de InterBase e indica que los datos de la tabla deben residir en un fichero externo al principal de la base de datos. Aunque el formato de este fichero no es ASCII, es relativamente sencillo de comprender y puede utili-

zarse para importar y exportar datos de un modo fácil entre InterBase y otras aplicaciones. En lo sucesivo no haremos uso de esta cláusula.

Para crear una tabla tenemos que definir columnas y restricciones sobre los valores que pueden tomar estas columnas. La forma más sencilla de definición de columna es la que sigue:

```
NombreColumna TipoDeDato
```

Por ejemplo:

```
create table Empleados (  
    Codigo          integer,  
    Nombre          varchar(30),  
    Contrato        date,  
    Salario         integer  
);
```

Columnas calculadas

Con InterBase tenemos la posibilidad de crear *columnas calculadas*, cuyos valores se derivan a partir de columnas existentes, sin necesidad de ser almacenados físicamente, para lo cual se utiliza la cláusula **computed by**. Aunque para este tipo de columnas podemos especificar explícitamente un tipo de datos, es innecesario, porque se puede deducir de la expresión que define la columna:

```
create table Empleados(  
    Codigo          integer,  
    Nombre          varchar,  
    Apellidos       varchar,  
    Salario         integer,  
    NombreCompleto computed by (Nombre || " " || Apellidos),  
    /* ... */  
);
```

El operador `||` sirve para concatenar cadenas de caracteres en InterBase.

En general, no es buena idea definir columnas calculadas en el servidor, sino que es preferible el uso de campos calculados en el cliente. Si utilizamos **computed by** hacemos que los valores de estas columnas viajen por la red con cada registro, aumentando el tráfico en la misma.

Valores por omisión

Otra posibilidad es la de definir valores por omisión para las columnas. Durante la inserción de filas, es posible no mencionar una determinada columna, en cuyo caso se le asigna a esta columna el valor por omisión. Si no se especifica algo diferente, el valor por omisión de SQL es **null**, el valor desconocido. Con la cláusula **default** cambiamos este valor:

```
Salario          integer default 0,
FechaContrato   date default "Now",
```

Observe en el ejemplo anterior el uso del literal "Now", para inicializar la columna con la fecha y hora actual en InterBase. En Oracle se utilizaría la función *sysdate*:

```
FechaContrato   date default sysdate
```

Si se mezclan las cláusulas **default** y **not null** en Oracle o InterBase, la primera debe ir antes de la segunda. En MS SQL Server, por el contrario, la cláusula **default** debe ir después de las especificaciones **null** ó **not null**:

```
FechaContrato   datetime not null default (getdate())
```

Restricciones de integridad

Durante la definición de una tabla podemos especificar condiciones que deben cumplirse para los valores almacenados en la misma. Por ejemplo, no nos basta saber que el salario de un empleado es un entero; hay que aclarar también que en circunstancias normales es también un entero positivo, y que no podemos dejar de especificar un salario a un trabajador. También nos puede interesar imponer condiciones más complejas, como que el salario de un empleado que lleva menos de un año con nosotros no puede sobrepasar cierta cantidad fija. En este epígrafe veremos cómo expresar estas *restricciones de integridad*.

La restricción más frecuente es pedir que el valor almacenado en una columna no pueda ser nulo. En el capítulo sobre manipulación de datos estudiaremos en profundidad este peculiar valor. El que una columna no pueda tener un valor nulo quiere decir que hay que suministrar un valor para esta columna durante la inserción de un nuevo registro, pero también que no se puede modificar posteriormente esta columna de modo que tenga un valor nulo. Esta restricción, como veremos dentro de poco, es indispensable para poder declarar claves primarias y claves alternativas. Por ejemplo:

```

create table Empleados(
    Codigo integer not null,
    Nombre varchar(30) not null,
    /* ... */
);

```

Cuando la condición que se quiere verificar es más compleja, se puede utilizar la cláusula **check**. Por ejemplo, la siguiente restricción verifica que los códigos de provincias se escriban en mayúsculas:

```

Provincia      varchar(2) check (Provincia = upper(Provincia))

```

Existen dos posibilidades con respecto a la ubicación de la mayoría de las restricciones: colocar la restricción a nivel de columna o a nivel de tabla. A nivel de columna, si la restricción afecta solamente a la columna en cuestión; a nivel de tabla si hay varias columnas involucradas. En mi humilde opinión, es más claro y legible expresar todas las restricciones a nivel de tabla, pero esto en definitiva es materia de gustos.

La cláusula **check** de InterBase permite incluso expresiones que involucran a otras tablas. Más adelante, al tratar la integridad referencial, veremos un ejemplo sencillo de esta técnica. Por el momento, analice la siguiente restricción, expresada a nivel de tabla:

```

create table Detalles (
    RefPedido      int not null,
    NumLinea       int not null,
    RefArticulo    int not null,
    Cantidad       int default 1 not null,
    Descuento      int default 0 not null,

    check (Descuento between 0 and 50 or "Marteens Corporation"=
        (select Nombre from Clientes
         where Codigo =
          (select RefCliente from Pedidos
           where Numero = Detalles.RefPedido))),

    /* ... */
);

```

Esta cláusula dice, en pocas palabras, que solamente el autor de este libro puede beneficiarse de descuentos superiores al 50%. ¡Algún privilegio tenía que corresponderme!

Claves primarias y alternativas

Las restricciones **check** nos permiten con relativa facilidad imponer condiciones sobre las filas de una tabla que pueden verificarse examinando solamente el registro activo. Cuando las reglas de consistencia involucran a varias filas a la vez, la expresión

de estas reglas puede complicarse bastante. En último caso, una combinación de cláusulas **check** y el uso de *triggers* o disparadores nos sirve para expresar *imperativamente* las reglas necesarias. Ahora bien, hay casos típicos de restricciones que afectan a varias filas a la vez que se pueden expresar *declarativamente*; estos casos incluyen a las restricciones de claves primarias y las de integridad referencial.

Mediante una clave primaria indicamos que una columna, o una combinación de columnas, debe tener valores únicos para cada fila. Por ejemplo, en una tabla de clientes, el código de cliente no debe repetirse en dos filas diferentes. Esto se expresa de la siguiente forma:

```
create table Clientes(
    Codigo integer not null primary key,
    Nombre varchar(30) not null,
    /* ... */
);
```

Si una columna pertenece a la clave primaria, debe estar especificada como no nula. Observe que en este caso hemos utilizado la restricción a nivel de columna. También es posible tener claves primarias compuestas, en cuyo caso la restricción hay que expresarla a nivel de tabla. Por ejemplo, en la tabla de detalles de pedidos, la clave primaria puede ser la combinación del número de pedido y el número de línea dentro de ese pedido:

```
create table Detalles(
    NumPedido integer not null,
    NumLinea integer not null,
    /* ... */
    primary key (NumPedido, NumLinea)
);
```

Solamente puede haber una clave primaria por cada tabla. De este modo, la clave primaria representa la *identidad* de los registros almacenados en una tabla: la información necesaria para localizar unívocamente un objeto. No es imprescindible especificar una clave primaria al crear tablas, pero es recomendable como método de trabajo.

Sin embargo, es posible especificar que otros grupos de columnas también poseen valores únicos dentro de las filas de una tabla. Estas restricciones son similares en sintaxis y semántica a las claves primarias, y utilizan la palabra reservada **unique**. En la jerga relacional, a estas columnas se le denominan *claves alternativas*. Una buena razón para tener claves alternativas puede ser que la columna designada como clave primaria sea en realidad una *clave artificial*. Se dice que una clave es artificial cuando no tiene un equivalente semántico en el sistema que se modela. Por ejemplo, el código de cliente no tiene una existencia *real*; nadie va por la calle con un 666 grabado en la frente. La verdadera clave de un cliente puede ser, además de su alma inmortal, su DNI. Pero el DNI debe almacenarse en una cadena de caracteres, y esto ocupa mucho más espacio que un código numérico. En este caso, el código numérico se utiliza

en las referencias a clientes, pues al tener menor tamaño la clave, pueden existir más entradas en un bloque de índice, y el acceso por índices es más eficiente. Entonces, la tabla de clientes puede definirse del siguiente modo:

```
create table Clientes (
    Codigo integer not null,
    DNI      varchar(9) not null,
    /* ... */
    primary key (Codigo),
    unique (DNI)
);
```

Por cada clave primaria o alternativa definida, InterBase crea un índice único para mantener la restricción. Este índice se bautiza según el patrón *rdbs\$primaryN*, donde *N* es un número único asignado por el sistema.

Integridad referencial

Un caso especial y frecuente de restricción de integridad es la conocida como restricción de *integridad referencial*. También se le denomina restricción por *clave externa* o *foránea* (*foreign key*). Esta restricción especifica que el valor almacenado en un grupo de columnas de una tabla debe encontrarse en los valores de las columnas en alguna fila de otra tabla, o de sí misma. Por ejemplo, en una tabla de pedidos se almacena el código del cliente que realiza el pedido. Este código debe corresponder al código de algún cliente almacenado en la tabla de clientes. La restricción puede expresarse de la siguiente manera:

```
create table Pedidos(
    Codigo      integer not null primary key,
    Cliente     integer not null references Clientes(Codigo),
    /* ... */
);
```

O, utilizando restricciones a nivel de tabla:

```
create table Pedidos(
    Codigo      integer not null,
    Cliente     integer not null,
    /* ... */
    primary key (Codigo),
    foreign key (Cliente) references Clientes(Codigo)
);
```

La columna o grupo de columnas a la que se hace referencia en la tabla maestra, la columna *Codigo* de *Clientes* en este caso, debe ser la clave primaria de esta tabla o ser una clave alternativa, esto es, debe haber sido definida una restricción **unique** sobre la misma.

Si todo lo que pretendemos es que no se pueda introducir una referencia a cliente inválida, se puede sustituir la restricción declarativa de integridad referencial por esta cláusula:

```
create table Pedidos(
  /* ... */
  check (Cliente in (select Codigo from Clientes))
);
```

La sintaxis de las expresiones será explicada en profundidad en el próximo capítulo, pero esta restricción **check** sencillamente comprueba que la referencia al cliente exista en la columna *Codigo* de la tabla *Clientes*.

Acciones referenciales

Sin embargo, las restricciones de integridad referencial ofrecen más que esta simple comprobación. Cuando tenemos una de estas restricciones, el sistema toma las riendas cuando tratamos de eliminar una fila maestra que tiene filas dependientes asociadas, y cuando tratamos de modificar la clave primaria de una fila maestra con las mismas condiciones. El estándar SQL-3 dicta una serie de posibilidades y reglas, denominadas *acciones referenciales*, que pueden aplicarse.

Lo más sencillo es prohibir estas operaciones, y es la solución que adoptan MS SQL Server (incluyendo la versión 7) y las versiones de InterBase anteriores a la 5. En la sintaxis más completa de SQL-3, esta política puede expresarse mediante la siguiente cláusula:

```
create table Pedidos(
  Codigo          integer not null primary key,
  Cliente         integer not null,
  /* ... */
  foreign key (Cliente) references Clientes(Codigo)
  on delete no action
  on update no action
);
```

Otra posibilidad es permitir que la acción sobre la tabla maestra se propague a las filas dependientes asociadas: eliminar un cliente puede provocar la desaparición de todos sus pedidos, y el cambio del código de un cliente modifica todas las referencias a este cliente. Por ejemplo:

```
create table Pedidos(
  Codigo          integer not null primary key,
  Cliente         integer not null
                  references Clientes(Codigo)
                  on delete no action on update cascade
  /* ... */
);
```

Observe que puede indicarse un comportamiento diferente para los borrados y para las actualizaciones.

En el caso de los borrados, puede indicarse que la eliminación de una fila maestra provoque que en la columna de referencia en las filas de detalles se asigne el valor nulo, o el valor por omisión de la columna de referencia:

```
insert into Empleados(Codigo, Nombre, Apellidos)
values(-1, "D'Arche", "Jeanne");

create table Pedidos(
    Codigo          integer not null primary key,
    Cliente         integer not null
                  references Clientes(Codigo)
                  on delete no action on update cascade,
    Empleado       integer default -1 not null
                  references Empleados(Codigo)
                  on delete set default on update cascade
/* ... */
);
```

InterBase 5 implementa todas estas estrategias, para lo cual necesita crear índices que le ayuden a verificar las restricciones de integridad referencial. La comprobación de la existencia de la referencia en la tabla maestra se realiza con facilidad, pues se trata en definitiva de una búsqueda en el índice único que ya ha sido creado para la gestión de la clave. Para prohibir o propagar los borrados y actualizaciones que afectarían a filas dependientes, la tabla que contiene la cláusula **foreign key** crea automáticamente un índice sobre las columnas que realizan la referencia. De este modo, cuando sucede una actualización en la tabla maestra, se pueden localizar con rapidez las posibles filas afectadas por la operación. Este índice nos ayuda en C++ Builder en la especificación de relaciones *master/detail* entre tablas. Los índices creados automáticamente para las relaciones de integridad referencial reciben nombres con el formato *rdB\$foreignN*, donde *N* es un número generado automáticamente.

Nombres para las restricciones

Cuando se define una restricción sobre una tabla, sea una verificación por condición o una clave primaria, alternativa o externa, es posible asignarle un nombre a la restricción. Este nombre es utilizado por InterBase en el mensaje de error que se produce al violarse la restricción, pero su uso fundamental es la manipulación posterior por parte de instrucciones como **alter table**, que estudiaremos en breve. Por ejemplo:

```

create table Empleados(
  /* ... */
  Salario          integer default 0,
  constraint      SalarioPositivo check(Salario >= 0)
  /* ... */
  constraint      NombreUnico
                   unique(Apellidos, Nombre)
);

```

También es posible utilizar nombres para las restricciones cuando éstas se expresan a nivel de columna. Las restricciones a las cuales no asignamos nombre reciben uno automáticamente por parte del sistema.

Definición y uso de dominios

SQL permite definir algo similar a los tipos de datos de los lenguajes tradicionales. Si estamos utilizando cierto tipo de datos con frecuencia, podemos definir un dominio para ese tipo de columna y utilizarlo consistentemente durante la definición del esquema de la base de datos. Un dominio, sin embargo, va más allá de la simple definición del tipo, pues permite expresar restricciones sobre la columna y valores por omisión. La sintaxis de una definición de dominio en InterBase es la siguiente:

```

create domain NombreDominio [as]
  TipoDeDato
  [ValorPorOmisión]
  [not null]
  [check(Condición)]
  [collate Criterio];

```

Cuando sobre un dominio se define una restricción de chequeo, no contamos con el nombre de la columna. Si antes expresábamos la restricción de que los códigos de provincia estuvieran en mayúsculas de esta forma:

```

Provincia          varchar(2) check(Provincia = upper(Provincia))

```

ahora necesitamos la palabra reservada **value** para referirnos al nombre de la columna:

```

create domain CodProv as
  varchar(2)
  check(value = upper(value));

```

El dominio definido, *CodProv*, puede utilizarse ahora para definir columnas:

```

Provincia          CodProv

```

Las cláusulas **check** de las definiciones de dominio no pueden hacer referencia a columnas de otras tablas.

Es aconsejable definir dominios en InterBase por una razón adicional: el Diccionario de Datos de C++ Builder los reconoce y asocia automáticamente a conjuntos de atributos (*attribute sets*). De esta forma, se ahorra mucho tiempo en la configuración de los objetos de acceso a campos.

Creación de índices

Como ya he explicado, InterBase crea índices de forma automática para mantener las restricciones de clave primaria, unicidad y de integridad referencial. En la mayoría de los casos, estos índices bastan para que el sistema funcione eficientemente. No obstante, es necesario en ocasiones definir índices sobre otras columnas. Esta decisión depende de la frecuencia con que se realicen consultas según valores almacenados en estas columnas, o de la posibilidad de pedir que una tabla se ordene de acuerdo al valor de las mismas. Por ejemplo, en la tabla de empleados es sensato pensar que el usuario de la base de datos deseará ver a los empleados listados por orden alfabético, o que querrá realizar búsquedas según un nombre y unos apellidos.

La sintaxis para crear un índice es la siguiente:

```
create [unique] [asc[ending] | desc[ending]] index Índice
on Tabla (Columna [, Columna ...])
```

Por ejemplo, para crear un índice sobre los apellidos y el nombre de los empleados necesitamos la siguiente instrucción:

```
create index NombreEmpleado on Empleados(Apellidos, Nombre)
```

Los índices creados por InterBase son todos sensibles a mayúsculas y minúsculas, y todos son mantenidos por omisión. El concepto de índice definido por expresiones y con condición de filtro es ajeno a la filosofía de SQL; este tipo de índices no se adapta fácilmente a la optimización automática de consultas. InterBase no permite tampoco crear índices sobre columnas definidas con la cláusula **computed by**.

Aunque definamos índices descendentes sobre una tabla en una base de datos SQL, el Motor de Datos de Borland no lo utilizará para ordenar tablas. Exactamente lo que sucede es que el BDE no permite que una tabla (no una consulta) pueda estar ordenada descendentemente por alguna de sus columnas, aunque la tabla mencione un índice descendente en su propiedad *IndexName*. En tal caso, el orden que se establece utiliza las mismas columnas del índice, pero ascendentemente.

Hay otro problema relacionado con los índices de InterBase. Al parecer, estos índices solamente pueden recorrerse en un sentido. Si definimos un índice ascendente sobre determinada columna de una tabla, y realizamos una consulta sobre la tabla con los resultados ordenados descendientemente por el valor de esa columna, InterBase no podrá aprovechar el índice creado.

Modificación de tablas e índices

SQL nos permite ser sabios y humanos a la vez: podemos equivocarnos en el diseño de una tabla o de un índice, y corregir posteriormente nuestro disparate. Sin caer en el sentimentalismo filosófico, es bastante común que una vez terminado el diseño de una base de datos surja la necesidad de añadir nuevas columnas a las tablas para almacenar información imprevista, o que tengamos que modificar el tipo o las restricciones activas sobre una columna determinada.

La forma más simple de la instrucción de modificación de tablas es la que elimina una columna de la misma:

```
alter table Tabla drop Columna [, Columna ...]
```

También se puede eliminar una restricción si conocemos su nombre. Por ejemplo, esta instrucción puede originar graves disturbios sociales:

```
alter table Empleados drop constraint SalarioPositivo;
```

Se pueden añadir nuevas columnas o nuevas restricciones sobre una tabla existente:

```
alter table Empleados add EstadoCivil varchar(8);
alter table Empleados
  add check (EstadoCivil in ("Soltero", "Casado", "Polígamo"));
```

Para los índices existen también instrucciones de modificación. En este caso, el único parámetro que se puede configurar es si el índice está activo o no:

```
alter index Indice (active | inactive);
```

Si un índice está inactivo, las modificaciones realizadas sobre la tabla no se propagan al índice, por lo cual necesitan menos tiempo para su ejecución. Si va a efectuar una entrada masiva de datos, quizás sea conveniente desactivar algunos de los índices secundarios, para mejorar el rendimiento de la operación. Luego, al activar el índice, éste se reconstruye dando como resultado una estructura de datos perfectamente balanceada. Estas instrucciones pueden ejecutarse periódicamente, para garantizar índices con tiempo de acceso óptimo:

```
alter index NombreEmpleado inactive;  
alter index NombreEmpleado active;
```

Otra instrucción que puede mejorar el rendimiento del sistema y que está relacionada con los índices es **set statistics**. Este comando calcula las estadísticas de uso de las claves dentro de un índice. El valor obtenido, conocido como *selectividad* del índice, es utilizado por InterBase para elaborar el plan de implementación de consultas. Normalmente no hay que invocar a esta función explícitamente, pero si las estadísticas de uso del índice han variado mucho es quizás apropiado utilizar la instrucción:

```
set statistics index NombreEmpleado;
```

Por último, las instrucciones **drop** nos permiten borrar objetos definidos en la base de datos, tanto tablas como índices:

```
drop table Tabla;  
drop index Indice;
```

Creación de vistas

Uno de los recursos más potentes de SQL, y de las bases de datos relacionales en general, es la posibilidad de definir tablas “virtuales” a partir de los datos almacenados en tablas “físicas”. Para definir una de estas tablas virtuales hay que definir qué operaciones relacionales se aplican a qué tablas bases. Este tipo de tabla recibe el nombre de *vista*.

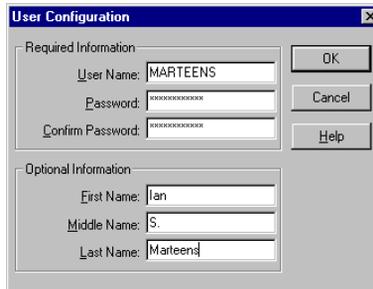
Como todavía no conocemos el lenguaje de consultas, que nos permite especificar las operaciones sobre tablas, postergaremos el estudio de las vistas para más adelante.

Creación de usuarios

InterBase soporta el concepto de usuarios a nivel del servidor, no de las bases de datos. Inicialmente, todos los servidores definen un único usuario especial: *SYSDBA*. Este usuario tiene los derechos necesarios para crear otros usuarios y asignarles contraseñas. Toda esa información se almacena en la base de datos *isc4.gdb*, que se instala automáticamente con InterBase. La gestión de los nombres de usuarios y sus contraseñas se realiza mediante la utilidad *Server Manager*.



Dentro de esta aplicación, hay que ejecutar el comando de menú *Tasks | User security*, para llegar al diálogo con el que podemos añadir, modificar o eliminar usuarios. La siguiente imagen muestra el diálogo de creación de nuevos usuarios:



El nombre del usuario *SYSDBA* no puede cambiarse, pero es casi una obligación cambiar su contraseña en cuanto termina la instalación de InterBase. Sin embargo, podemos eliminar al administrador de la lista de usuarios del sistema. Si esto sucede, ya no será posible añadir o modificar nuevos usuarios en ese servidor. Así que tenga cuidado con lo que hace.

El sistema de seguridad explicado tiene un par de aparentes "fallos". En primer lugar, cualquier usuario con acceso al disco duro puede sustituir el fichero *ise4.gdb* con uno suyo. Más grave aún: si copiamos el fichero *gdb* de la base de datos en un servidor en el cual conozcamos la contraseña del administrador, tendremos acceso total a los datos, aunque este acceso nos hubiera estado vedado en el servidor original.

En realidad, el fallo consiste en permitir que cualquier mequetrefe pueda acceder a nuestras apreciadas bases de datos. Así que, antes de planear la protección del sistema de gestión de base de datos (ya sea InterBase o cualquier otro), ocúpese de controlar el acceso al servidor de la gente indeseable.

Asignación de privilegios

Una vez creados los objetos de la base de datos, es necesario asignar derechos sobre los mismos a los demás usuarios. Inicialmente, el dueño de una tabla es el usuario que la crea, y tiene todos los derechos de acceso sobre la tabla. Los derechos de acceso indican qué operaciones pueden realizarse con la tabla. Naturalmente, los nombres de estos derechos o privilegios coinciden con los nombres de las operaciones correspondientes:

Privilegio	Operación
select	Lectura de datos
update	Modificación de datos existentes
insert	Creación de nuevos registros
delete	Eliminación de registros
all	Los cuatro privilegios anteriores
execute	Ejecución (para procedimientos almacenados)

La instrucción que otorga derechos sobre una tabla es la siguiente:

```
grant Privilegios on Tabla to Usuarios [with grant option]
```

Por ejemplo:

```
/* Derecho de sólo-lectura al público en general */
grant select on Articulos to public;
/* Todos los derechos a un par de usuarios */
grant all privileges on Clientes to Spade, Marlowe;
/* Monsieur Poirot sólo puede modificar salarios (¡qué peligro!) */
grant update(Salario) on Empleados to Poirot;
/* Privilegio de inserción y borrado, con opción de concesión */
grant insert, delete on Empleados to Vance with grant option;
```

He mostrado unas cuantas posibilidades de la instrucción. En primer lugar, podemos utilizar la palabra clave **public** cuando queremos conceder ciertos derechos a todos los usuarios. En caso contrario, podemos especificar uno o más usuarios como destinatarios del privilegio. Luego, podemos ver que el privilegio **update** puede llevar entre paréntesis la lista de columnas que pueden ser modificadas. Por último, vemos que a Mr. Philo Vance no solamente le permiten contratar y despedir empleados, sino que también, gracias a la cláusula **with grant option**, puede conceder estos derechos a otros usuarios, aún no siendo el creador de la tabla. Esta opción debe utilizarse con cuidado, pues puede provocar una propagación descontrolada de privilegios entre usuarios indeseables.

¿Y qué pasa si otorgamos privilegios y luego nos arrepentimos? No hay problema, pues para esto tenemos la instrucción **revoke**:

```
revoke [grant option for] Privilegios on Tabla from Usuarios
```

Hay que tener cuidado con los privilegios asignados al público. La siguiente instrucción no afecta a los privilegios de Sam Spade sobre la tabla de artículos, porque antes se le ha concedido al público en general el derecho de lectura sobre la misma:

```
/* Spade se ríe de este ridículo intento */  
revoke all on Articulos from Spade;
```

Existen variantes de las instrucciones **grant** y **revoke** pensadas para asignar y retirar privilegios sobre tablas a procedimientos almacenados, y para asignar y retirar derechos de ejecución de procedimientos a usuarios.

Roles

Los roles son una especificación del SQL-3 que InterBase 5 ha implementado. Si los usuarios se almacenan y administran a nivel de servidor, los roles, en cambio, se definen a nivel de cada base de datos. De este modo, podemos trasladar con más facilidad una base de datos desarrollada en determinado servidor, con sus usuarios particulares, a otro servidor, en el cual existe históricamente otro conjunto de usuarios.

El sujeto de la validación por contraseña sigue siendo el usuario. La relación entre usuarios y roles es la siguiente: un usuario puede *asumir* un rol al conectarse a la base de datos; actualmente, InterBase no permite asumir roles después de este momento. A un rol se le pueden otorgar privilegios exactamente igual que a un usuario, utilizando **grant** y **revoke**. Cuando un usuario asume un rol, los privilegios del rol se suman a los privilegios que se le han concedido como usuario. Por supuesto, un usuario debe contar con la autorización para asumir un rol, lo cual se hace también mediante **grant** y **revoke**.

¿Un ejemplo? Primero necesitamos crear los roles adecuados en la base de datos:

```
create role Domador;  
create role Payaso;  
create role Mago;
```

Ahora debemos asignar los permisos sobre tablas y otros objetos a los roles. Esto no impide que, en general, se puedan también asignar permisos específicos a usuarios puntuales:

```
grant all privileges on Animales to Domador, Mago;  
grant select on Animales to Payaso;
```

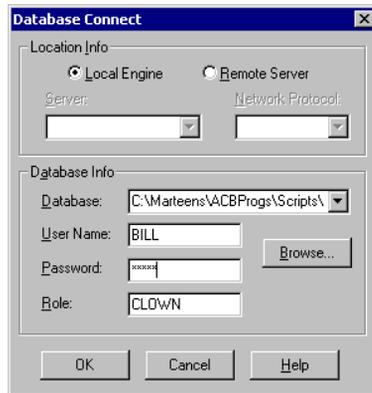
Hasta aquí no hemos mencionado a los usuarios, por lo que los resultados de estas instrucciones son válidos de servidor a servidor. Finalmente, debemos asignar los

usuarios en sus respectivos roles, y esta operación sí depende del conjunto de usuarios de un servidor:

```
grant Payaso to Bill, Steve, RonaldMcDonald;  
grant Domador to Ian with admin option;
```

La opción **with admin option** me permite asignar el rol de domador a otros usuarios. De este modo, siempre habrá quien se ocupe de los animales cuando me ausente del circo por vacaciones.

La pregunta importante es: ¿cómo puede el usuario indicar a InterBase que desea asumir determinado rol en una conexión? Si nos conectamos con las herramientas del propio InterBase, comprobaremos que existe un cuadro de edición que nos pregunta por el rol a asumir. Por supuesto, podemos dejarlo vacío, si nos queremos atener exclusivamente a nuestros privilegios como usuarios:



Pero si la conexión la realizamos desde C++ Builder tendremos que programar un poco. Los roles se añaden al soporte de InterBase del BDE en la versión 5.0.1.23. El BDE que acompañaba a Delphi 4, por ejemplo, no contaba con esta posibilidad. El nuevo SQL Link de InterBase introduce el parámetro *ROLE NAME*, y es aquí donde debemos indicar el rol del usuario. Lamentablemente, el diálogo de conexión del componente *TDatabase* tampoco considera a los roles, por lo que si necesitamos que el usuario dicte dinámicamente los parámetros de conexión tendremos que interceptar el evento *OnLogin* del componente *TDatabase*.

Un ejemplo completo de *script* SQL

Incluyo a continuación un ejemplo completo de *script* SQL con la definición de tablas e índices para una sencilla aplicación de entrada de pedidos. En un capítulo posterior,

ampliaremos este *script* para incluir *triggers*, generadores y procedimientos almacenados que ayuden a expresar las reglas de empresa de la base de datos.

```

create database "C:\Pedidos\Pedidos.GDB"
user "SYSDBA" password "masterkey"
page_size 2048;

/* Creación de las tablas */

create table Clientes (
    Codigo          int not null,
    Nombre          varchar(30) not null,
    Direccion1     varchar(30),
    Direccion2     varchar(30),
    Telefono       varchar(15),
    UltimoPedido   date default "Now",

    primary key    (Codigo)
);

create table Empleados (
    Codigo          int not null,
    Apellidos      varchar(20) not null,
    Nombre         varchar(15) not null,
    FechaContrato  date default "Now",
    Salario        int,
    NombreCompleto computed by (Nombre || " " || Apellidos),

    primary key    (Codigo)
);

create table Articulos (
    Codigo          int not null,
    Descripcion     varchar(30) not null,
    Existencias    int default 0,
    Pedidos        int default 0,
    Costo          int,
    PVP            int,

    primary key    (Codigo)
);

create table Pedidos (
    Numero         int not null,
    RefCliente     int not null,
    RefEmpleado    int,
    FechaVenta     date default "Now",
    Total          int default 0,

    primary key    (Numero),
    foreign key    (RefCliente) references Clientes (Codigo)
                    on delete no action on update cascade
);

create table Detalles (
    RefPedido      int not null,
    NumLinea       int not null,

```

```
RefArticulo    int not null,  
Cantidad       int default 1 not null,  
Descuento      int default 0 not null  
check (Descuento between 0 and 100),  
  
primary key    (RefPedido, NumLinea),  
foreign key    (RefPedido) references Pedidos (Numero),  
               on delete cascade on update cascade  
foreign key    (RefArticulo) references Articulos (Codigo)  
               on delete no action on update cascade  
);  
  
/* Indices secundarios */  
  
create index NombreCliente on Clientes(Nombre);  
create index NombreEmpleado on Empleados(Apellidos, Nombre);  
create index Descripcion on Articulos(Descripcion);  
  
/***** FIN DEL SCRIPT *****/
```

Consultas y modificaciones

DESDE SU MISMO ORIGEN, la definición del modelo relacional de Codd incluía la necesidad de un lenguaje para realizar consultas *ad-hoc*. Debido a la forma particular de representación de datos utilizada por este modelo, el tener relaciones o tablas y no contar con un lenguaje de alto nivel para reintegrar los datos almacenados es más bien una maldición que una bendición. Es asombroso, por lo tanto, cuánto tiempo vivió el mundo de la programación sobre PCs sin poder contar con SQL o algún mecanismo similar. Aún hoy, cuando un programador de Clipper o de COBOL comienza a trabajar en C++ Builder, se sorprende de las posibilidades que le abre el uso de un lenguaje de consultas integrado dentro de sus aplicaciones.

La instrucción **select**, del Lenguaje de Manipulación de Datos de SQL, nos permite consultar la información almacenada en una base de datos relacional. La sintaxis y posibilidades de esta sola instrucción son tan amplias y complicadas como para merecer un capítulo para ella solamente. En este mismo capítulo estudiaremos las posibilidades de las instrucciones **update**, **insert** y **delete**, que permiten la modificación del contenido de las tablas de una base de datos.

Para los ejemplos de este capítulo utilizaré la base de datos *mastsql.gdb* que viene con los ejemplos de C++ Builder, en el siguiente directorio:

```
C:\Archivos de programa\Archivos comunes\Borland Shared\Data
```

Estas tablas también se encuentran en formato Paradox, en el mismo subdirectorio. Puede utilizar el programa *Database Desktop* para probar el uso de SQL sobre tablas Paradox y dBase. Sin embargo, trataré de no tocar las peculiaridades del Motor de SQL Local ahora, dejando esto para los capítulos 24 y 27, que explican cómo utilizar SQL desde C++ Builder.

La instrucción **select**: el lenguaje de consultas

A grandes rasgos, la estructura de la instrucción **select** es la siguiente:

```

select [distinct] lista-de-expresiones
from lista-de-tablas
[where condición-de-selección]
[group by lista-de-columnas]
[having condición-de-selección-de-grupos]
[order by lista-de-columnas]
[union instrucción-de-selección]

```

¿Qué se supone que “hace” una instrucción **select**? Esta es la pregunta del millón: una instrucción **select**, en principio, no “hace” sino que “define”. La instrucción define un conjunto virtual de filas y columnas, o más claramente, define una tabla virtual. Qué se hace con esta “tabla virtual” es ya otra cosa, y depende de la aplicación que le estemos dando. Si estamos en un intérprete que funciona en modo texto, puede ser que la ejecución de un **select** se materialice mostrando en pantalla los resultados, página a página, o quizás en salvar el resultado en un fichero de texto. En C++ Builder, las instrucciones **select** se utilizan para “alimentar” un componente denominado *TQuery*, al cual se le puede dar casi el mismo uso que a una tabla “real”, almacenada físicamente.

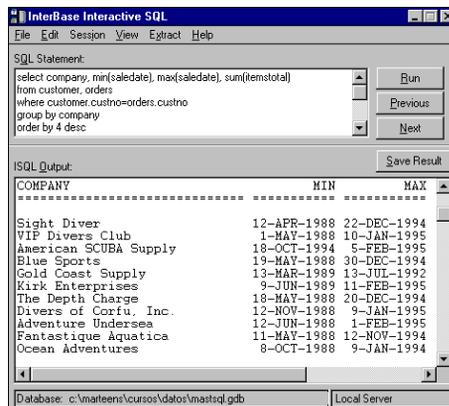
A pesar de la multitud de secciones de una selección completa, el formato básico de la misma es muy sencillo, y se reduce a las tres primeras secciones:

```

select lista-de-expresiones
from lista-de-tablas
[where condición-de-selección]

```

La cláusula **from** indica de dónde se extrae la información de la consulta, en la cláusula **where** opcional se dice qué filas deseamos en el resultado, y con **select** especificamos los campos o expresiones de estas filas que queremos obtener. Muchas veces se dice que la cláusula **where** limita la tabla “a lo largo”, pues elimina filas de la misma, mientras que la cláusula **select** es una selección “horizontal”.



La condición de selección

La forma más simple de instrucción **select** es la que extrae el conjunto de filas de una sola tabla que satisfacen cierta condición. Por ejemplo:

```
select *
from Customer
where State = "HI"
```

Esta consulta simple debe devolver todos los datos de los clientes ubicados en Hawaii. El asterisco que sigue a la cláusula **select** es una alternativa a listar todos los nombres de columna de la tabla que se encuentra en la cláusula **from**.

En este caso hemos utilizado una simple igualdad. La condición de búsqueda de la cláusula **where** admite los seis operadores de comparación (=, <>, <, >, <=, >=) y la creación de condiciones compuestas mediante el uso de los operadores lógicos **and**, **or** y **not**. La prioridad entre estos tres es la misma que en C. No hace falta encerrar las comparaciones entre paréntesis, porque incluso **not** se evalúa después de cualquier comparación:

```
select *
from Customer
where State = "HI"
and LastInvoiceDate > "1/1/1993"
```

Observe cómo la constante de fecha puede escribirse como si fuera una cadena de caracteres.

Operadores de cadenas

Además de las comparaciones usuales, necesitamos operaciones más sofisticadas para trabajar con las cadenas de caracteres. Uno de los operadores admitidos por SQL estándar es el operador **like**, que nos permite averiguar si una cadena satisface o no cierto patrón de caracteres. El segundo operando de este operador es el patrón, una cadena de caracteres, dentro de la cual podemos incluir los siguientes comodines:

Carácter	Significado
%	Cero o más caracteres arbitrarios.
_(subrayado)	Un carácter cualquiera.

No vaya a pensar que el comodín % funciona como el asterisco en los nombres de ficheros de MS-DOS; SQL es malo, pero no tanto. Después de colocar un asterisco en un nombre de fichero, MS-DOS ignora cualquier otro carácter que escribamos a continuación, mientras que **like** sí los tiene en cuenta. También es diferente el com-

portamiento del subrayado con respecto al signo de interrogación de DOS: en el intérprete de comandos de este sistema operativo significa cero o un caracteres, mientras que en SQL significa exactamente un carácter.

Expresión	Cadena aceptada	Cadena no aceptada
Customer like '% Ocean'	'Pacific Ocean'	'Ocean Paradise'
Fruta like 'Manzana_'	'Manzanas'	'Manzana'

También es posible aplicar funciones para extraer o modificar información de una cadena de caracteres; el repertorio de funciones disponibles depende del sistema de bases de datos con el que se trabaje. Por ejemplo, el intérprete SQL para tablas locales de C++ Builder acepta las funciones **upper**, **lower**, **trim** y **substring** de SQL estándar. Esta última función tiene una sintaxis curiosa. Por ejemplo, para extraer las tres primeras letras de una cadena se utiliza la siguiente expresión:

```
select substring(Nombre from 1 for 3)
from Empleados
```

Si estamos trabajando con InterBase, podemos aumentar el repertorio de funciones utilizando *funciones definidas por el usuario*. En el capítulo 11 mostraremos cómo.

Yo sólo quiero los diez primeros...

El resultado de una sentencia **select** puede contener un número impredecible de registros, pero en muchos casos a usted solamente le interesa un puñado de filas representativas del resultado. Por ejemplo: queremos veinte clientes cualesquiera de Hawaii. O, si calculamos las ventas por cada cliente y ordenamos el resultado de acuerdo a este valor, puede que necesitemos sólo los cinco clientes que más han comprado.

Lamentablemente, no existe un estándar en SQL para expresar la condición anterior, y cada sistema que implementa algo parecido lo hace a su aire. InterBase, en particular, no ofrece operadores para limitar el tamaño de un conjunto resultado. Más adelante, al estudiar los procedimientos almacenados, veremos cómo superar esta limitación.

Microsoft SQL Server ofrece una cláusula en la sentencia **select** para elegir un grupo inicial de filas:

```
select top 20 *
from Clientes
where Estado = 'HI'
```

Incluso nos deja recuperar un porcentaje del conjunto resultado:

```
select top 20 percent *
from Clientes
where Estado = 'HI'
```

En el caso anterior, no se trata de los veinte primeros clientes, sino de la quinta parte de la cantidad total de clientes.

DB2 pone a nuestra disposición un mecanismo similar:

```
select *
from Clientes
where Estado = 'HI'
fetch first 20 rows only
```

Por mucho que busquemos en los lenguajes de programación modernos, nunca veremos aparecer dos palabras reservadas consecutivamente. Aquí estamos viendo dos palabras claves juntas antes y después del número entero, como si estuvieran escoltándolo. Para colmo de la verbosidad, sepa usted que también podíamos haber escrito **row** en singular si quisiéramos únicamente la primera fila.

Oracle, por su parte, dispone de una pseudo columna, *rownum*, que puede utilizarse del siguiente modo:

```
select *
from Clientes
where Estado = 'HI' and
rownum <= 20
```

El valor nulo: enfrentándonos a lo desconocido

La edad de una persona es un valor no negativo, casi siempre menor de 969 años, que es la edad a la que dicen que llegó Matusalén. Puede ser un entero igual a 1, 20, 40 ... o no conocerse. Se puede “resolver” este problema utilizando algún valor especial para indicar el valor desconocido, digamos -1. Claro, el valor especial escogido no debe formar parte del dominio posible de valores. Por ejemplo, en el archivo de Urgencias de un hospital americano, *John Doe* es un posible valor para los pacientes no identificados.

¿Y qué pasa si no podemos prescindir de valor alguno dentro del rango? Porque John Doe es un nombre raro, pero posible. ¿Y qué pasaría si se intentan operaciones con valores desconocidos? Por ejemplo, para representar un envío cuyo peso se desconoce se utiliza el valor -1, un peso claramente imposible excepto para entes como Kate Moss. Luego alguien pregunta a la base de datos cuál es el peso total de los envíos de un período dado. Si en ese período se realizaron dos envíos, uno de 25 kilogramos y otro de peso desconocido, la respuesta errónea será un peso total de 24

kilogramos. Es evidente que la respuesta debería ser, simplemente, “peso total desconocido”.

La solución de SQL es introducir un nuevo valor, **null**, que pertenece a cualquier dominio de datos, para representar la información desconocida. La regla principal que hay que conocer cuando se trata con valores nulos es que cualquier expresión, aparte de las expresiones lógicas, en la que uno de sus operandos tenga el valor nulo se evalúa automáticamente a nulo. Esto es: nulo más veinticinco vale nulo, ¿de acuerdo?

Cuando se trata de evaluar expresiones lógicas en las cuales uno de los operandos puede ser nulo las cosas se complican un poco, pues hay que utilizar una lógica de tres valores. De todos modos, las reglas son intuitivas. Una proposición falsa en conjunción con cualquier otra da lugar a una proposición falsa; una proposición verdadera en disyunción con cualquier otra da lugar a una proposición verdadera. La siguiente tabla resume las reglas del uso del valor nulo en expresiones lógicas:

AND	false	null	true
false	false	false	false
null	false	null	null
true	false	null	true

OR	false	null	true
false	false	null	true
null	null	null	true
true	true	true	true

Por último, si lo que desea es saber si el valor de un campo es nulo o no, debe utilizar el operador **is null**:

```
select *
from Events
where Event_Description is null
```

La negación de este operador es el operador **is not null**, con la negación en medio. Esta sintaxis no es la usual en lenguajes de programación, pero se suponía que SQL debía parecerse lo más posible al idioma inglés.

Eliminación de duplicados

Normalmente, no solemos guardar filas duplicadas en una tabla, por razones obvias. Pero es bastante frecuente que el resultado de una consulta contenga filas duplicadas. El operador **distinct** se puede utilizar, en la cláusula **select**, para corregir esta situación. Por ejemplo, si queremos conocer en qué ciudades residen nuestros clientes podemos preguntar lo siguiente:

```
select City
from Customer
```

Pero en este caso obtenemos 55 ciudades, algunas de ellas duplicadas. Para obtener las 47 diferentes ciudades de la base de datos tecleamos:

```
select distinct City
from Customer
```

Productos cartesianos y encuentros

Como para casi todas las cosas, la gran virtud del modelo relacional es, a la vez, su mayor debilidad. Me refiero a que cualquier modelo del “mundo real” puede representarse atomizándolo en relaciones: objetos matemáticos simples y predecibles, de fácil implementación en un ordenador (jaquellos ficheros *dbfs*...!). Para reconstruir el modelo original, en cambio, necesitamos una operación conocida como “encuentro natural” (*natural join*).

Comencemos con algo más sencillo: con los productos cartesianos. Un producto cartesiano es una operación matemática entre conjuntos, la cual produce todas las parejas posibles de elementos, perteneciendo el primer elemento de la pareja al primer conjunto, y el segundo elemento de la pareja al segundo conjunto. Esta es la operación habitual que efectuamos mentalmente cuando nos ofrecen el menú en un restaurante. Los dos conjuntos son el de los primeros platos y el de los segundos platos. Desde la ventana de la habitación donde escribo puedo ver el menú del mesón de la esquina:

Primer plato	Segundo plato
Macarrones a la boloñesa	Escalope a la milanesa
Judías verdes con jamón	Pollo a la parrilla
Crema de champiñones	Chuletas de cordero

Si *PrimerPlato* y *SegundoPlato* fuesen tablas de una base de datos, la instrucción

```
select *
from PrimerPlato, SegundoPlato
```

devolvería el siguiente conjunto de filas:

Primer plato	Segundo plato
Macarrones a la boloñesa	Escalope a la milanesa
Macarrones a la boloñesa	Pollo a la parrilla
Macarrones a la boloñesa	Chuletas de cordero
Judías verdes con jamón	Escalope a la milanesa
Judías verdes con jamón	Pollo a la parrilla
Judías verdes con jamón	Chuletas de cordero

Primer plato	Segundo plato
Crema de champiñones	Escalope a la milanesa
Crema de champiñones	Pollo a la parrilla
Crema de champiñones	Chuletas de cordero

Es fácil ver que, incluso con tablas pequeñas, el tamaño del resultado de un producto cartesiano es enorme. Si a este ejemplo “real” le añadimos el hecho también “real” de que el mismo mesón ofrece al menos tres tipos diferentes de postres, elegir nuestro menú significa seleccionar entre 27 posibilidades distintas. Por eso siempre pido un café solo al terminar con el segundo plato.

Claro está, no todas las combinaciones de platos hacen una buena comida. Pero para eso tenemos la cláusula **where**: para eliminar aquellas combinaciones que no satisfacen ciertos criterios. ¿Volvemos al mundo de las facturas y órdenes de compra? En la base de datos *dbdemos*, la información sobre pedidos está en la tabla *orders*, mientras que la información sobre clientes se encuentra en *customer*. Queremos obtener la lista de clientes y sus totales por pedidos. Estupendo, pero los totales de pedidos están en la tabla *orders*, en el campo *ItemsTotal*, y en esta tabla sólo tenemos el código del cliente, en el campo *CustNo*. Los nombres de clientes se encuentran en el campo *Company* de la tabla *customer*, donde además volvemos a encontrar el código de cliente, *CustNo*. Así que partimos de un producto cartesiano entre las dos tablas, en el cual mostramos los nombres de clientes y los totales de pedidos:

```
select Company, ItemsTotal
from Customer, Orders
```

Como tenemos unos 55 clientes y 205 pedidos, esta inocente consulta genera unas 11275 filas. La última vez que hice algo así fue siendo estudiante, en el centro de cálculos de mi universidad, para demostrarle a una profesora de Filosofía lo ocupado que estaba en ese momento.

En realidad, de esas 11275 filas nos sobran unas 11070, pues solamente son válidas las combinaciones en las que coinciden los códigos de cliente. La instrucción que necesitamos es:

```
select Company, ItemsTotal
from Customer, Orders
where Customer.CustNo = Orders.CustNo
```

Esto es un *encuentro natural*, un producto cartesiano restringido mediante la igualdad de los valores de dos columnas de las tablas básicas.

El ejemplo anterior ilustra también un punto importante: cuando queremos utilizar en la instrucción el nombre de los campos *ItemsTotal* y *Company* los escribimos tal y

como son. Sin embargo, cuando utilizamos *CustNo* hay que aclarar a qué tabla original nos estamos refiriendo. Esta técnica se conoce como *calificación de campos*.

¿Un ejemplo más complejo? Suponga que desea añadir el nombre del empleado que recibió el pedido. La tabla *orders* tiene un campo *EmpNo* para el código del empleado, mientras que la información sobre empleados se encuentra en la tabla *employee*. La instrucción necesaria es una simple ampliación de la anterior:

```
select Company, ItemsTotal, FirstName || " " || LastName
from Customer, Orders, Employee
where Customer.CustNo = Orders.CustNo
       and Orders.EmpNo = Employee.EmpNo
```

Con 42 empleados en la base de datos de ejemplo y sin las restricciones de la cláusula **where**, hubiéramos obtenido un resultado de 473550 filas.

Ordenando los resultados

Una de las garantías de SQL es que podemos contar con que el compilador SQL genere automáticamente, o casi, el mejor código posible para evaluar las instrucciones. Esto también significa que, en el caso general, no podemos predecir con completa seguridad cuál será la estrategia utilizada para esta evaluación. Por ejemplo, en la instrucción anterior no sabemos si el compilador va a recorrer cada fila de la tabla de clientes para encontrar las filas correspondientes de pedidos o empleados, o si resultará más ventajoso recorrer las filas de pedidos para recuperar los nombres de clientes y empleados. Esto quiere decir, en particular, que no sabemos en qué orden se nos van a presentar las filas. En mi ordenador, utilizando Database Desktop sobre las tablas originales en formato Paradox, parece ser que se recorren primeramente las filas de la tabla de empleados.

¿Qué hacemos si el resultado debe ordenarse por el nombre de compañía? Para esto contamos con la cláusula **order by**, que se sitúa siempre al final de la consulta. En este caso, ordenamos por nombre de compañía el resultado con la instrucción:

```
select Company, ItemsTotal, FirstName || " " || LastName
from Customer, Orders, Employee
where Customer.CustNo = Orders.CustNo
       and Orders.EmpNo = Employee.EmpNo
order by Company
```

No se puede ordenar por una fila que no existe en el resultado de la instrucción. Si quisiéramos que los pedidos de cada compañía se ordenaran, a su vez, por la fecha de venta, habría que añadir el campo *SaleDate* al resultado y modificar la cláusula de ordenación del siguiente modo:

```
select Company, ItemsTotal, SaleDate, FirstName || " " || LastName
from Customer, Orders, Employee
where Customer.CustNo = Orders.CustNo
and Orders.EmpNo = Employee.EmpNo
order by Company, SalesDate desc
```

Con la opción **desc** obtenemos los registros por orden descendente de fechas: primero los más recientes. Existe una opción **asc**, para cuando queremos enfatizar el sentido ascendente de una ordenación. Generalmente no se usa, pues es lo que asume el compilador.

Otra posibilidad de la cláusula **order by** es utilizar números en vez de nombres de columnas. Esto es necesario si se utilizan expresiones en la cláusula **select** y se quiere ordenar por dicha expresión. Por ejemplo:

```
select OrderNo, SalesDate, ItemsTotal - AmountPaid
from Orders
order by 3 desc
```

No se debe abusar de los números de columnas, pues esta técnica puede desaparecer en SQL-3 y hace menos legible la consulta. Una forma alternativa de ordenar por columnas calculadas es utilizar sinónimos para las columnas:

```
select OrderNo, SalesDate, ItemsTotal - AmountPaid as Diferencia
from Orders
order by Diferencia desc
```

El uso de grupos

Ahora queremos sumar todos los totales de los pedidos para cada compañía, y ordenar el resultado por este total de forma descendente, para obtener una especie de *ranking* de las compañías según su volumen de compras. Esto es, hay que agrupar todas las filas de cada compañía y mostrar la suma de cierta columna dentro de cada uno de esos grupos.

Para producir grupos de filas en SQL se utiliza la cláusula **group by**. Cuando esta cláusula está presente en una consulta, va situada inmediatamente después de la cláusula **where**, o de la cláusula **from** si no se han efectuado restricciones. En nuestro caso, la instrucción con la cláusula de agrupamiento debe ser la siguiente:

```
select Company, sum(ItemsTotal)
from Customer, Orders
where Customer.CustNo = Orders.OrderNo
group by Company
order by 2 desc
```

Observe la forma en la cual se le ha aplicado la función **sum** a la columna *ItemsTotal*. Aunque pueda parecer engorroso el diseño de una consulta con grupos, hay una regla muy fácil que simplifica los razonamientos: en la cláusula **select** solamente pueden aparecer columnas especificadas en la cláusula **group by**, o funciones estadísticas aplicadas a cualquier otra expresión. *Company*, en este ejemplo, puede aparecer directamente porque es la columna por la cual se está agrupando. Si quisiéramos obtener además el nombre de la persona de contacto en la empresa, el campo *Contact* de la tabla *customer*, habría que agregar esta columna a la cláusula de agrupación:

```
select Company, Contact, sum(ItemsTotal)
from Customer, Orders
where Customer.CustNo = Orders.OrderNo
group by Company, Contact
order by 2 desc
```

En realidad, la adición de *Contact* es redundante, pues *Company* es única dentro de la tabla *customer*, pero eso lo sabemos nosotros, no el compilador de SQL. Sin embargo, InterBase puede optimizar mejor la consulta anterior si la planteamos del siguiente modo:

```
select Company, max(Contact), sum(ItemsTotal)
from Customer, Orders
where Customer.CustNo = Orders.OrderNo
group by Company
order by 2 desc
```

Si agrupamos por el nombre de compañía, dentro de cada grupo todas las personas de contacto serán iguales. Por lo tanto, nos da lo mismo mostrar el máximo o el mínimo de todos esos nombres, y así eliminamos una columna de la cláusula **group by**.

Funciones de conjuntos

Existen cinco funciones de conjuntos en SQL, conocidas en inglés como *aggregate functions*. Estas funciones son:

Función	Significado
count	Cantidad de valores no nulos en el grupo
min	El valor mínimo de la columna dentro del grupo
max	El valor máximo de la columna dentro del grupo
sum	La suma de los valores de la columna dentro del grupo
avg	El promedio de la columna dentro del grupo

Por supuesto, no toda función es aplicable a cualquier tipo de columna. Las sumas, por ejemplo, solamente valen para columnas de tipo numérico. Hay otros detalles curiosos relacionados con estas funciones, como que los valores nulos son ignorados

por todas, o que se puede utilizar un asterisco como parámetro de la función **count**. En este último caso, se calcula el número de filas del grupo. Así que no apueste a que la siguiente consulta dé siempre dos valores idénticos, si es posible que la columna involucrada contenga valores nulos:

```
select avg(Columna), sum(Columna)/count(*)
from Tabla
```

En el ejemplo se muestra la posibilidad de utilizar funciones de conjuntos sin utilizar grupos. En esta situación se considera que toda la tabla constituye un único grupo. Es también posible utilizar el operador **distinct** como prefijo del argumento de una de estas funciones:

```
select Company, count(distinct EmpNo)
from Customer, Orders
where Customer.CustNo = Orders.CustNo
```

La consulta anterior muestra el número de empleados que han atendido los pedidos de cada compañía.

Oracle añade un par de funciones a las que ya hemos mencionado: **variance** y **stddev**, para la varianza y la desviación estándar. En realidad, estas funciones pueden calcularse a partir de las anteriores. Por ejemplo, la varianza de la columna *x* de la tabla *Tabla* puede obtenerse mediante la siguiente instrucción:

```
select (sum(x*x) - sum(x) * sum(x) / count(*)) / (count(*) - 1)
from Tabla
```

En cuanto a la desviación estándar, basta con extraer la raíz cuadrada de la varianza. Es cierto que InterBase no tiene una función predefinida para las raíces cuadradas, pero es fácil implementarla mediante una función de usuario.

La cláusula having

Según lo que hemos explicado hasta el momento, en una instrucción **select** se evalúa primeramente la cláusula **from**, que indica qué tablas participan en la consulta, luego se eliminan las filas que no satisfacen la cláusula **where** y, si hay un **group by** por medio, se agrupan las filas resultantes. Hay una posibilidad adicional: después de agrupar se pueden descartar filas consolidadas de acuerdo a otra condición, esta vez expresada en una cláusula **having**. En la parte **having** de la consulta solamente pueden aparecer columnas agrupadas o funciones estadísticas aplicadas al resto de las columnas. Por ejemplo:

```

select Company
from Customer, Orders
where Customer.CustNo = Orders.CustNo
group by Company
having count(*) > 1

```

La consulta anterior muestra las compañías que han realizado más de un pedido. Es importante darse cuenta de que no podemos modificar esta consulta para que nos muestre las compañías que *no* han realizado todavía pedidos.

Una regla importante de optimización: si en la cláusula **having** existen condiciones que implican solamente a las columnas mencionadas en la cláusula **group by**, estas condiciones deben moverse a la cláusula **where**. Por ejemplo, si queremos eliminar de la consulta utilizada como ejemplo a las compañías cuyo nombre termina con las siglas 'S.L.' debemos hacerlo en **where**, no en **group by**. ¿Para qué esperar a agrupar para eliminar filas que podían haberse descartado antes? Aunque muchos compiladores realizan esta optimización automáticamente, es mejor no fiarse.

El uso de sinónimos para tablas

Es posible utilizar dos o más veces una misma tabla en una misma consulta. Si hacemos esto tendremos que utilizar *sinónimos* para distinguir entre los distintos usos de la tabla en cuestión. Esto será necesario al calificar los campos que utilizemos. Un sinónimo es simplemente un nombre que colocamos a continuación del nombre de una tabla en la cláusula **from**, y que en adelante se usa como sustituto del nombre de la tabla.

Por ejemplo, si quisiéramos averiguar si hemos introducido por error dos veces a la misma compañía en la tabla de clientes, pudiéramos utilizar la instrucción:

```

select distinct C1.Company
from Customer C1, Customer C2
where C1.CustNo < C2.CustNo
and C1.Company = C2.Company

```

En esta consulta *C1* y *C2* se utilizan como sinónimos de la primera y segunda aparición, respectivamente, de la tabla *customer*. La lógica de la consulta es sencilla. Buscamos todos los pares que comparten el mismo nombre de compañía y eliminamos aquellos que tienen el mismo código de compañía. Pero en vez de utilizar una desigualdad en la comparación de códigos, utilizamos el operador “menor que”, para eliminar la aparición de pares dobles en el resultado previo a la aplicación del operador **distinct**. Estamos aprovechando, por supuesto, la unicidad del campo *CustNo*.

La siguiente consulta muestra otro caso en que una tabla aparece dos veces en una cláusula **from**. En esta ocasión, la base de datos es *iblocal*, el ejemplo InterBase que viene con C++ Builder. Queremos mostrar los empleados junto con los jefes de sus departamentos:

```
select e2.full_name, e1.full_name
from   employee e1, department d, employee e2
where  d.dept_no = e1.dept_no
       and d.mngr_no = e2.emp_no
       and e1.emp_no <> e2.emp_no
order  by 1, 2
```

Aquellos lectores que hayan trabajado en algún momento con lenguajes xBase reconocerán en los sinónimos SQL un mecanismo similar al de los “alias” de xBase. C++ Builder utiliza, además, los sinónimos de tablas en el intérprete de SQL local cuando el nombre de la tabla contiene espacios en blanco o el nombre de un directorio.

Subconsultas: selección única

Si nos piden el total vendido a una compañía determinada, digamos a Ocean Paradise, podemos resolverlo ejecutando dos instrucciones diferentes. En la primera obtenemos el código de Ocean Paradise:

```
select Customer.CustNo
from   Customer
where  Customer.Company = "Ocean Paradise"
```

El código buscado es, supongamos, 1510. Con este valor en la mano, ejecutamos la siguiente instrucción:

```
select sum(Orders.ItemsTotal)
from   Orders
where  Orders.CustNo = 1510
```

Incómodo, ¿no es cierto? La alternativa es utilizar la primera instrucción como una expresión dentro de la segunda, del siguiente modo:

```
select sum(Orders.ItemsTotal)
from   Orders
where  Orders.CustNo = (
       select Customer.CustNo
       from   Customer
       where  Customer.Company = "Ocean Paradise")
```

Para que la subconsulta anterior pueda funcionar correctamente, estamos asumiendo que el conjunto de datos retornado por la subconsulta produce una sola fila. Esta es, realmente, una apuesta arriesgada. Puede fallar por dos motivos diferentes: puede que la subconsulta no devuelva ningún valor o puede que devuelva más de uno. Si no

se devuelve ningún valor, se considera que la subconsulta devuelve el valor **null**. Si devuelve dos o más valores, el intérprete produce un error.

A este tipo de subconsulta que debe retornar un solo valor se le denomina *selección única*, en inglés, *singleton select*. Las selecciones únicas también pueden utilizarse con otros operadores de comparación, además de la igualdad. Así por ejemplo, la siguiente consulta retorna información sobre los empleados contratados después de Pedro Pérez:

```
select *
from Employee E1
where E1.HireDate > (
  select E2.HireDate
  from Employee E2
  where E2.FirstName = "Pedro"
  and E2.LastName = "Pérez")
```

Si está preguntándose acerca de la posibilidad de cambiar el orden de los operandos, ni lo sueñe. La sintaxis de SQL es muy rígida, y no permite este tipo de virtuosismos.

Subconsultas: los operadores *in* y *exists*

En el ejemplo anterior garantizábamos la singularidad de la subconsulta gracias a la cláusula **where**, que especificaba una búsqueda sobre una clave única. Sin embargo, también se pueden aprovechar las situaciones en que una subconsulta devuelve un conjunto de valores. En este caso, el operador a utilizar cambia. Por ejemplo, si queremos los pedidos correspondientes a las compañías en cuyo nombre figura la palabra Ocean, podemos utilizar la instrucción:

```
select *
from Orders
where Orders.CustNo in (
  select Customer.CustNo
  from Customer
  where upper(Customer.Company) like "%OCEAN%")
```

El nuevo operador es el operador **in**, y la expresión es verdadera si el operando izquierdo se encuentra en la lista de valores retornada por la subconsulta. Esta consulta puede descomponerse en dos fases. Durante la primera fase se evalúa el segundo **select**:

```
select Customer.CustNo
from Customer
where upper(Customer.Company) like "%OCEAN%"
```

El resultado de esta consulta consiste en una serie de códigos: aquellos que corresponden a las compañías con Ocean en su nombre. Supongamos que estos códigos

sean 1510 (Ocean Paradise) y 5515 (Ocean Adventures). Entonces puede ejecutarse la segunda fase de la consulta, con la siguiente instrucción, equivalente a la original:

```
select *
from Orders
where Orders.OrderNo in (1510, 5515)
```

Este otro ejemplo utiliza la negación del operador **in**. Si queremos las compañías que no nos han comprado nada, hay que utilizar la siguiente consulta:

```
select *
from Customer
where Customer.CustNo not in (
    select Orders.CustNo
    from Orders)
```

Otra forma de plantearse las consultas anteriores es utilizando el operador **exists**. Este operador se aplica a una subconsulta y devuelve verdadero en cuanto localiza una fila que satisface las condiciones de la instrucción **select**. El primer ejemplo de este epígrafe puede escribirse de este modo:

```
select *
from Orders
where exists (
    select *
    from Customer
    where upper(Customer.Company) like "%OCEAN%"
    and Orders.CustNo = Customer.CustNo)
```

Observe el asterisco en la cláusula **select** de la subconsulta. Como lo que nos interesa es saber si existen filas que satisfacen la expresión, nos da lo mismo qué valor se está retornando. El segundo ejemplo del operador **in** se convierte en lo siguiente al utilizar **exists**:

```
select *
from Customer
where not exists (
    select *
    from Orders
    where Orders.CustNo = Customer.CustNo)
```

Subconsultas correlacionadas

Preste atención al siguiente detalle: la última subconsulta del epígrafe anterior tiene una referencia a una columna perteneciente a la tabla definida en la cláusula **from** más externa. Esto quiere decir que no podemos explicar el funcionamiento de la instrucción dividiéndola en dos fases, como con las selecciones únicas: la ejecución de la subconsulta y la simplificación de la instrucción externa. En este caso, para cada

fila retornada por la cláusula **from** externa, la tabla *customer*, hay que volver a evaluar la subconsulta teniendo en cuenta los valores actuales: los de la columna *CustNo* de la tabla de clientes. A este tipo de subconsultas se les denomina, en el mundo de la programación SQL, *subconsultas correlacionadas*.

Si hay que mostrar los clientes que han pagado algún pedido contra reembolso (en inglés, *COD*, o *cash on delivery*), podemos realizar la siguiente consulta con una subselección correlacionada:

```
select *
from Customer
where 'COD' in (
    select distinct PaymentMethod
    from Orders
    where Orders.CustNo = Customer.CustNo)
```

En esta instrucción, para cada cliente se evalúan los pedidos realizados por el mismo, y se muestra el cliente solamente si dentro del conjunto de métodos de pago está la cadena '*COD*'. El operador **distinct** de la subconsulta es redundante, pero nos ayuda a entenderla mejor.

Otra subconsulta correlacionada: queremos los clientes que no han comprado nada aún. Ya vimos como hacerlo utilizando el operador **not in** ó el operador **not exists**. Una alternativa es la siguiente:

```
select *
from Customer
where 0 = (
    select count(*)
    from Orders
    where Orders.CustNo = Customer.CustNo)
```

Sin embargo, utilizando SQL Local esta consulta es más lenta que las otras dos soluciones. La mayor importancia del concepto de subconsulta correlacionada tiene que ver con el hecho de que algunos sistemas de bases de datos limitan las actualizaciones a vistas definidas con instrucciones que contienen subconsultas de este tipo.

Equivalencias de subconsultas

En realidad, las subconsultas son un método para aumentar la expresividad del lenguaje SQL, pero no son imprescindibles. Con esto quiero decir que muchas consultas se formulan de modo más natural si se utilizan subconsultas, pero que existen otras consultas equivalentes que no hacen uso de este recurso. La importancia de esta equivalencia reside en que el intérprete de SQL Local de Delphi 1 no permitía subconsultas. También sucede que la versión 4 de InterBase no optimizaba correctamente ciertas subconsultas.

Un problema relacionado es que, aunque un buen compilador de SQL debe poder identificar las equivalencias y evaluar la consulta de la forma más eficiente, en la práctica el utilizar ciertas construcciones sintácticas puede dar mejor resultado que utilizar otras equivalentes, de acuerdo al compilador que empleemos.

Veamos algunas equivalencias. Teníamos una consulta, en el epígrafe sobre selecciones únicas, que mostraba el total de compras de Ocean Paradise:

```
select sum(Orders.ItemsTotal)
from Orders
where Orders.CustNo = (
    select Customer.CustNo
    from Customer
    where Customer.Company = "Ocean Paradise")
```

Esta consulta es equivalente a la siguiente:

```
select sum(ItemsTotal)
from Customer, Orders
where Customer.Company = "Ocean Paradise"
and Customer.CustNo = Orders.OrderNo
```

Aquella otra consulta que mostraba los pedidos de las compañías en cuyo nombre figuraba la palabra “Ocean”:

```
select *
from Orders
where Orders.CustNo in (
    select Customer.CustNo
    from Customer
    where upper(Customer.Company) like "%OCEAN%")
```

es equivalente a esta otra:

```
select *
from Customer, Orders
where upper(Customer.Company) like "%OCEAN%"
and Customer.CustNo = Orders.CustNo
```

Para esta consulta en particular, ya habíamos visto una consulta equivalente que hacía uso del operador **exists**; en este caso, es realmente más difícil de entender la consulta con **exists** que su equivalente sin subconsultas.

La consulta correlacionada que buscaba los clientes que en algún pedido habían pagado contra reembolso:

```
select *
from Customer
where 'COD' in (
```

```

select distinct PaymentMethod
from Orders
where Orders.CustNo = Customer.CustNo)

```

puede escribirse, en primer lugar, mediante una subconsulta no correlacionada:

```

select *
from Customer
where Customer.CustNo in (
select Orders.CustNo
from Orders
where PaymentMethod = 'COD' )

```

pero también se puede expresar en forma “plana”:

```

select distinct Customer.CustNo, Customer.Company
from Customer, Orders
where Customer.CustNo = Orders.CustNo
and Orders.PaymentMethod = 'COD'

```

Por el contrario, las consultas que utilizan el operador **not in** y, por lo tanto sus equivalentes con **not exists**, no tienen equivalente plano, con lo que sabemos hasta el momento. Para poder aplanarlas hay que utilizar *encuentros externos*.

Encuentros externos

El problema de los encuentros naturales es que cuando relacionamos dos tablas, digamos *customer* y *orders*, solamente mostramos las filas que tienen una columna en común. No hay forma de mostrar los clientes que *no* tienen un pedido con su código ... y solamente esos. En realidad, se puede utilizar la operación de *diferencia* entre conjuntos para lograr este objetivo, como veremos en breve. Se pueden evaluar todos los clientes, y a ese conjunto restarle el de los clientes que sí tienen pedidos. Pero esta operación, por lo general, se implementa de forma menos eficiente que la alternativa que mostraremos a continuación.

¿Cómo funciona un encuentro natural? Una posible implementación consistiría en recorrer mediante un bucle la primera tabla, supongamos que sea *customer*. Para cada fila de esa tabla tomaríamos su columna *CustNo* y buscaríamos, posiblemente con un índice, las filas correspondientes de *orders* que contengan ese mismo valor en la columna del mismo nombre. ¿Qué pasa si no hay ninguna fila en *orders* que satisfaga esta condición? Si se trata de un encuentro natural, común y corriente, no se muestran los datos de ese cliente. Pero si se trata de la extensión de esta operación, conocida como *encuentro externo* (*outer join*), se muestra aunque sea una vez la fila correspondiente al cliente. Un encuentro muestra, sin embargo, pares de filas, ¿qué valores podemos esperar en la fila de pedidos? En ese caso, se considera que todas las columnas de la tabla de pedidos tienen valores nulos. Si tuviéramos estas dos tablas:

Customers	
<i>CustNo</i>	<i>Company</i>
1510	Ocean Paradise
1666	Marteens' Diving Academy

Orders	
<i>OrderNo</i>	<i>CustNo</i>
1025	1510
1139	1510

el resultado de un encuentro externo como el que hemos descrito, de acuerdo a la columna *CustNo*, sería el siguiente:

Customer.CustNo	Company	OrderNo	Orders.CustNo
1510	Ocean Paradise	1025	1510
1510	Ocean Paradise	1139	1510
1666	Marteens' Diving Academy	null	null

Con este resultado en la mano, es fácil descubrir quién es el tacaño que no nos ha pedido nada todavía, dejando solamente las filas que tengan valores nulos para alguna de las columnas de la segunda tabla.

Este encuentro externo que hemos explicado es, en realidad, un encuentro externo *por la izquierda*, pues la primera tabla tendrá todas sus filas en el resultado final, aunque no exista fila correspondiente en la segunda. Naturalmente, también existe un encuentro externo *por la derecha* y un encuentro externo *simétrico*.

El problema de este tipo de operaciones es que su inclusión en SQL fue bastante tardía. Esto trajo como consecuencia que distintos fabricantes utilizaran sintaxis propias para la operación. En el estándar ANSI para SQL del año 87 no hay referencias a esta instrucción, pero sí la hay en el estándar del 92. Utilizando esta sintaxis, que es la permitida por el SQL local, la consulta que queremos se escribe del siguiente modo:

```
select *
from Customer left outer join Orders
on Customer.CustNo = Orders.CustNo
where Orders.OrderNo is null
```

Observe que se ha extendido la sintaxis de la cláusula **from**.

El encuentro externo por la izquierda puede escribirse en Oracle de esta forma alternativa:

```
select *
from Customer, Orders
where Customer.CustNo (+) = Orders.CustNo
and Orders.OrderNo is null
```

La mayoría de las aplicaciones de los encuentros externos están relacionadas con la generación de informes. Pongamos por caso que tenemos una tabla de clientes y una tabla relacionada de teléfonos, asumiendo que un cliente puede tener asociado un número de teléfono, varios o ninguno. Si queremos listar los clientes y sus números de teléfono y utilizamos un encuentro natural, aquellos clientes de los que desconocemos el teléfono no aparecerán en el listado. Es necesario entonces recurrir a un encuentro externo por la izquierda.

La curiosa sintaxis del encuentro interno

De la misma manera en que hay una sintaxis especial para los encuentros externos, existe una forma equivalente de expresar el encuentro “normal”, o interno:

```
select Company, OrderNo
from Customer inner join Orders
on Customer.CustNo = Orders.CustNo
```

Fácilmente se comprende que la consulta anterior es equivalente a:

```
select Company, OrderNo
from Customer, Orders
where Customer.CustNo = Orders.CustNo
```

¿Por qué nos complican la vida con la sintaxis especial los señores del comité ANSI? El propósito de las cláusulas **inner** y **outer join** es la definición de “expresiones de tablas” limitadas. Hasta el momento, nuestra sintaxis solamente permitía nombres de tabla en la cláusula **from**, pero gracias a las nuevas cláusulas se pueden escribir consultas más complejas.

```
select Company, OrderNo, max(Discount)
from Customer C
left outer join
(Orders O inner join Items I
on O.OrderNo = I.OrderNo)
on C.CustNo = O.CustNo
group by Company, OrderNo
```

Si hubiéramos omitido los paréntesis en la cláusula **from** de la consulta anterior hubiéramos perdido las filas de clientes que no han realizado pedidos. También pueden combinarse entre sí varios encuentros externos.

Aunque no soy partidario de utilizar la complicada sintaxis del **inner join**, le conviene familiarizarse con la misma, pues varias herramientas de C++ Builder (SQL Builder, el editor de consultas de Decision Cube) insistirán tozudamente en modificar sus encuentros “naturales” para que usen esta notación.

Las instrucciones de actualización

Son tres las instrucciones de actualización de datos reconocidas en SQL: **delete**, **update** e **insert**. Estas instrucciones tienen una sintaxis relativamente simple y están limitadas, en el sentido de que solamente cambian datos en una tabla a la vez. La más sencilla de las tres es **delete**, la instrucción de borrado:

```
delete from Tabla
where Condición
```

La instrucción elimina de la tabla indicada todas las filas que se ajustan a cierta condición. En dependencia del sistema de bases de datos de que se trate, la condición de la cláusula **where** debe cumplir ciertas restricciones. Por ejemplo, aunque InterBase admite la presencia de subconsultas en la condición de selección, otros sistemas no lo permiten.

La segunda instrucción, **update**, nos sirve para modificar las filas de una tabla que satisfacen cierta condición:

```
update Tabla
set Columna = Valor [, Columna = Valor ...]
where Condición
```

Al igual que sucede con la instrucción **delete**, las posibilidades de esta instrucción dependen del sistema que la implementa. InterBase, en particular, permite actualizar columnas con valores extraídos de otras tablas; para esto utilizamos subconsultas en la cláusula **set**:

```
update Customer
set LastInvoiceDate =
    (select max(SaleDate) from Orders
     where Orders.CustNo = Customer.CustNo)
```

Por último, tenemos la instrucción **insert**, de la cual tenemos dos variantes. La primera permite insertar un solo registro, con valores constantes:

```
insert into Tabla [ (Columnas) ]
values (Valores)
```

La lista de columnas es opcional; si se omite, se asume que la instrucción utiliza todas las columnas en orden de definición. En cualquier caso, el número de columnas empleado debe coincidir con el número de valores. El objetivo de todo esto es que si no se indica un valor para alguna columna, el valor de la columna en el nuevo registro se inicializa con el valor definido por omisión; recuerde que si en la definición de la tabla no se ha indicado nada, el valor por omisión es **null**:

```

insert into Employee(EmpNo, LastName, FirstName)
values (666, "Bonaparte", "Napoleón")
/* El resto de las columnas, incluida la del salario, son nulas */

```

La segunda variante de **insert** permite utilizar como fuente de valores una expresión **select**:

```

insert into Tabla [ (Columnas) ]
InstrucciónSelect

```

Esta segunda variante no estuvo disponible en el intérprete de SQL local hasta la versión 4 del BDE. Se utiliza con frecuencia para copiar datos de una tabla a otra:

```

insert into Resumen(Empresa, TotalVentas)
select Company, sum(ItemsTotal)
from Customer, Orders
where Customer.CustNo = Orders.CustNo
group by Company

```

La semántica de la instrucción update

Según el estándar ANSI para SQL, durante la ejecución de la cláusula **set** de una instrucción **update** en la que se modifican varias columnas, se evalúan primero todos los valores a asignar y se guardan en variables temporales, y luego se realizan todas las asignaciones. De este modo, el orden de las asignaciones no afecta el resultado final.

Supongamos que tenemos una tabla llamada *Tabla*, con dos columnas del mismo tipo: *A* y *B*. Para intercambiar los valores de ambas columnas puede utilizarse la siguiente instrucción:

```

update Tabla
set    A = B, B = A

```

Y realmente, así suceden las cosas en Oracle y MS SQL Server. ... pero no en InterBase. Si la instrucción anterior se ejecuta en InterBase, al final de la misma los valores de la columna *B* se han copiado en la columna *A*, pero no al contrario. Al menos en la versión 5.5 del producto.

Este es un detalle a tener en cuenta por los programadores que intentan transportar una aplicación de una plataforma a otra. Imagino que en algún nivel del estándar se deje a la decisión del fabricante qué semántica implementar en la cláusula **set**. Y es que el ANSI SQL es demasiado permisivo para mi gusto.

Vistas

Se puede aprovechar una instrucción **select** de forma tal que el conjunto de datos que define se pueda utilizar “casi” como una tabla real. Para esto, debemos definir una *vista*. La instrucción necesaria tiene la siguiente sintaxis:

```
create view NombreVista[Columnas]
as InstrucciónSelect
[with check option]
```

Por ejemplo, podemos crear una vista para trabajar con los clientes que viven en Hawaii:

```
create view Hawaiianos as
select *
from Customer
where State = "HI"
```

A partir del momento en que se ejecuta esta instrucción, el usuario de la base de datos se encuentra con una nueva tabla, *Hawaiianos*, con la cual puede realizar las mismas operaciones que realizaba con las tablas “físicas”. Puede utilizar la nueva tabla en una instrucción **select**:

```
select *
from Hawaiianos
where LastInvoiceDate >=
(select avg(LastInvoiceDate) from Customer)
```

En esta vista en particular, puede también eliminar insertar o actualizar registros:

```
delete from Hawaiianos
where LastInvoiceDate is null;

insert into Hawaiianos(CustNo, Company, State)
values (8888, "Ian Marteens' Diving Academy", "HI")
```

No todas las vistas permiten operaciones de actualización. Las condiciones que deben cumplirse para ser actualizables, además, dependen del sistema de bases de datos en que se definan. Los sistemas más restrictivos exigen que la instrucción **select** tenga una sola tabla en la cláusula **from**, que no contenga consultas anidadas y que no haga uso de operadores tales como **group by**, **distinct**, etc.

Cuando una vista permite actualizaciones se nos plantea el problema de qué hacer si se inserta un registro que no pertenece lógicamente a la vista. Por ejemplo, ¿podríamos insertar dentro de la vista *Hawaiianos* una empresa con sede social en la ciudad

costera de Cantalapedra⁵? Si permitiésemos esto, después de la inserción el registro recién insertado “desaparecería” inmediatamente de la vista (aunque no de la tabla base, *Customer*). El mismo conflicto se produciría al actualizar la columna *State* de un hawaiano.

Para controlar este comportamiento, SQL define la cláusula **with check option**. Si se especifica esta opción, no se permiten inserciones ni modificaciones que violen la condición de selección impuesta a la vista; si intentamos una operación tal, se produce un error de ejecución. Por el contrario, si no se incluye la opción en la definición de la vista, estas operaciones se permiten, pero nos encontraremos con situaciones como las descritas, en que un registro recién insertado o modificado desaparece misteriosamente por no pertenecer a la vista.

⁵ Cuando escribí la primera versión de este libro, no sabía que había un Cantalapedra en España; pensé que nombre tan improbable era invento mío. Mis disculpas a los cantalapedrenses, pues me parece que viven en ciudad sin costas.

Procedimientos almacenados y triggers

CON ESTE CAPÍTULO COMPLETAMOS la presentación de los sublenguajes de SQL, mostrando el lenguaje de definición de procedimientos de InterBase. Desgraciadamente, los lenguajes de procedimientos de los distintos sistemas de bases de datos difieren entre ellos, al no existir todavía un estándar al respecto. De los dialectos existentes, he elegido nuevamente InterBase por dos razones. La primera, y fundamental, es que es el sistema SQL que *usted* tiene a mano (asumiendo que tiene C++ Builder). La segunda es que el dialecto de InterBase para procedimientos es el que más se asemeja al propuesto en el borrador del estándar SQL-3. De cualquier manera, las diferencias entre dialectos no son demasiadas, y no le costará mucho trabajo entender el lenguaje de procedimientos de cualquier otro sistema de bases de datos.

¿Para qué usar procedimientos almacenados?

Un *procedimiento almacenado* (*stored procedure*) es, sencillamente, un algoritmo cuya definición reside en la base de datos, y que es ejecutado por el servidor del sistema. Aunque SQL-3 define formalmente un lenguaje de programación para procedimientos almacenados, cada uno de los sistemas de bases de datos importantes a nivel comercial implementa su propio lenguaje para estos recursos. InterBase ofrece un dialecto parecido a la propuesta de SQL-3; Oracle tiene un lenguaje llamado PL-SQL; Microsoft SQL Server ofrece el denominado Transact-SQL. No obstante, las diferencias entre estos lenguajes son mínimas, principalmente sintácticas, siendo casi idénticas las capacidades expresivas.

El uso de procedimientos almacenados ofrece las siguientes ventajas:

- Los procedimientos almacenados ayudan a mantener la consistencia de la base de datos.

Las instrucciones básicas de actualización, **update**, **insert** y **delete**, pueden combinarse arbitrariamente si dejamos que el usuario tenga acceso ilimitado a las mismas. No toda combinación de actualizaciones cumplirá con las reglas de consistencia de la base de datos. Hemos visto que algunas de estas reglas se pueden expresar declarativamente durante la definición del esquema relacional. El mejor ejemplo son las restricciones de integridad referencial. Pero, ¿cómo expresar declarativamente que para cada artículo presente en un pedido, debe existir un registro correspondiente en la tabla de movimientos de un almacén? Una posible solución es prohibir el uso directo de las instrucciones de actualización, revocando permisos de acceso al público, y permitir la modificación de datos solamente a partir de procedimientos almacenados.

- Los procedimientos almacenados permiten superar las limitaciones del lenguaje de consultas.

SQL no es un lenguaje completo. Un problema típico en que falla es en la definición de *clausuras relacionales*. Tomemos como ejemplo una tabla con dos columnas: *Objeto* y *Parte*. Esta tabla contiene pares como los siguientes:

Objeto	Parte
Cuerpo humano	Cabeza
Cuerpo humano	Tronco
Cabeza	Ojos
Cabeza	Boca
Boca	Dientes

¿Puede el lector indicar una consulta que liste todas las partes incluidas en la cabeza? Lo que falla es la posibilidad de expresar algoritmos recursivos. Para resolver esta situación, los procedimientos almacenados pueden implementarse de forma tal que devuelvan conjuntos de datos, en vez de valores escalares. En el cuerpo de estos procedimientos se pueden realizar, entonces, llamadas recursivas.

- Los procedimientos almacenados pueden reducir el tráfico en la red.

Un procedimiento almacenado se ejecuta en el servidor, que es precisamente donde se encuentran los datos. Por lo tanto, no tenemos que explorar una tabla de arriba a abajo desde un ordenador cliente para extraer el promedio de ventas por empleado durante el mes pasado. Además, por regla general el servidor es una máquina más potente que las estaciones de trabajo, por lo que puede que ahorremos tiempo de ejecución para una petición de infor-

mación. No conviene, sin embargo, abusar de esta última posibilidad, porque una de las ventajas de una red consiste en distribuir el tiempo de procesador.

- Con los procedimientos almacenados se puede ahorrar tiempo de desarrollo.

Siempre que existe una información, a alguien se le puede ocurrir un nuevo modo de aprovecharla. En un entorno cliente/servidor es típico que varias aplicaciones diferentes trabajen con las mismas bases de datos. Si centralizamos en la propia base de datos la imposición de las reglas de consistencia, no tendremos que volverlas a programar de una aplicación a otra. Además, evitamos los riesgos de una mala codificación de estas reglas, con la consiguiente pérdida de consistencia.

Como todas las cosas de esta vida, los procedimientos almacenados también tienen sus inconvenientes. Ya he mencionado uno de ellos: si se centraliza todo el tratamiento de las reglas de consistencia en el servidor, corremos el riesgo de saturar los procesadores del mismo. El otro inconveniente es la poca portabilidad de las definiciones de procedimientos almacenados entre distintos sistemas de bases de datos. Si hemos desarrollado procedimientos almacenados en InterBase y queremos migrar nuestra base de datos a Oracle (o viceversa), estaremos obligados a partir “casi” de cero; algo se puede aprovechar, de todos modos.

Cómo se utiliza un procedimiento almacenado

Un procedimiento almacenado puede utilizarse desde una aplicación cliente, desarrollada en cualquier lenguaje de programación que pueda acceder a la interfaz de programación de la base de datos, o desde las propias utilidades interactivas del sistema. En la VCL tenemos el componente *TStoredProc*, diseñado para la ejecución de estos procedimientos. En un capítulo posterior veremos cómo suministrar parámetros, ejecutar procedimientos y recibir información utilizando este componente.

En el caso de InterBase, también es posible ejecutar un procedimiento almacenado directamente desde la aplicación *Windows ISQL*, mediante la siguiente instrucción:

```
execute procedure NombreProcedimiento [ListaParámetros];
```

La misma instrucción puede utilizarse en el lenguaje de definición de procedimientos y *triggers* para llamar a un procedimiento dentro de la definición de otro. Es posible también definir procedimientos recursivos. InterBase permite hasta un máximo de 1000 llamadas recursivas por procedimiento.

El carácter de terminación

Los procedimientos almacenados de InterBase deben necesariamente escribirse en un fichero *script* de SQL. Más tarde, este fichero debe ser ejecutado desde la utilidad *Windows ISQL* para que los procedimientos sean incorporados a la base de datos. Hemos visto las reglas generales del uso de *scripts* en InterBase en el capítulo de introducción a SQL. Ahora tenemos que estudiar una característica de estos *scripts* que anteriormente hemos tratado superficialmente: el carácter de terminación.

Por regla general, cada instrucción presente en un *script* es leída y ejecutada de forma individual y secuencial. Esto quiere decir que el intérprete de *scripts* lee del fichero hasta que detecta el fin de instrucción, ejecuta la instrucción recuperada, y sigue así hasta llegar al final del mismo. El problema es que este proceso de extracción de instrucciones independientes se basa en la detección de un carácter especial de terminación. Por omisión, este carácter es el punto y coma; el lector habrá observado que todos los ejemplos de instrucciones SQL que deben colocarse en *scripts* han sido, hasta el momento, terminados con este carácter.

Ahora bien, al tratar con el lenguaje de procedimientos y *triggers* encontraremos instrucciones y cláusulas que deben terminar con puntos y comas. Si el intérprete de *scripts* tropieza con uno de estos puntos y comas pensará que se encuentra frente al fin de la instrucción, e intentará ejecutar lo que ha leído hasta el momento; casi siempre, una instrucción incompleta. Por lo tanto, debemos cambiar el carácter de terminación de *Windows ISQL* cuando estamos definiendo *triggers* o procedimientos almacenados. La instrucción que nos ayuda para esto es la siguiente:

```
set term Terminador
```

Como carácter de terminación podemos escoger cualquier carácter o combinación de los mismos lo suficientemente rara como para que no aparezca dentro de una instrucción del lenguaje de procedimientos. Por ejemplo, podemos utilizar el acento circunflejo:

```
set term ^;
```

Observe cómo la instrucción que cambia el carácter de terminación debe terminar ella misma con el carácter antiguo. Al finalizar la creación de todos los procedimientos que necesitamos, debemos restaurar el antiguo carácter de terminación:

```
set term ;^
```

En lo sucesivo asumiremos que el carácter de terminación ha sido cambiado al acento circunflejo.

Procedimientos almacenados en InterBase

La sintaxis para la creación de un procedimiento almacenado en InterBase es la siguiente:

```
create procedure Nombre
  [ ( ParámetrosDeEntrada ) ]
  [ returns ( ParámetrosDeSalida ) ]
as CuerpoDeProcedimiento
```

Las cláusulas *ParámetrosDeEntrada* y *ParámetrosDeSalida* representan listas de declaraciones de parámetros. Los parámetros de salida pueden ser más de uno; esto significa que el procedimiento almacenado que retorna valores no se utiliza como si fuese una función de un lenguaje de programación tradicional. El siguiente es un ejemplo de cabecera de procedimiento:

```
create procedure TotalPiezas(PiezaPrincipal char(15))
  returns (Total integer)
as
/* ... Aquí va el cuerpo ... */
```

El cuerpo del procedimiento, a su vez, se divide en dos secciones, siendo opcional la primera de ellas: la sección de declaración de variables locales, y una instrucción compuesta, **begin...end**, que agrupa las instrucciones del procedimiento. Las variables se declaran en este verboso estilo, *á la 1970*:

```
declare variable V1 integer;
declare variable V2 char(50);
```

Estas son las instrucciones permitidas por los procedimientos almacenados de InterBase:

- Asignaciones:

```
Variable = Expresión
```

Las variables pueden ser las declaradas en el propio procedimiento, parámetros de entrada o parámetros de salida.

- Llamadas a procedimientos:

```
execute procedure NombreProc [ParsEntrada]
  [returning_values ParsSalida]
```

No se admiten expresiones en los parámetros de entrada; mucho menos en los de salida.

- Condicionales:

```
if (Condición) then Instrucción [else Instrucción]
```

- Bucles controlados por condiciones:

```
while (Condición) do Instrucción
```

- Instrucciones SQL:

Cualquier instrucción de manipulación, **insert**, **update** ó **delete**, puede incluirse en un procedimiento almacenado. Estas instrucciones pueden utilizar variables locales y parámetros, siempre que estas variables estén precedidas de dos puntos, para distinguirlas de los nombres de columnas. Por ejemplo, si *Minimo* y *Aumento* son variables o parámetros, puede ejecutarse la siguiente instrucción:

```
update Empleados
set Salario = Salario * :Aumento
where Salario < :Minimo;
```

Se permite el uso directo de instrucciones **select** si devuelven una sola fila; para consultas más generales se utiliza la instrucción **for** que veremos dentro de poco. Estas selecciones únicas van acompañadas por una cláusula **into** para transferir valores a variables o parámetros:

```
select Empresa
from Clientes
where Codigo = 1984
into :NombreEmpresa;
```

- Iteración sobre consultas:

```
for InstrucciónSelect into Variables do Instrucción
```

Esta instrucción recorre el conjunto de filas definido por la instrucción **select**. Para cada fila, transfiere los valores a las variables de la cláusula **into**, de forma similar a lo que sucede con las selecciones únicas, y ejecuta entonces la instrucción de la sección **do**.

- Lanzamiento de excepciones:

```
exception NombreDeExcepción
```

Similar a la instrucción **throw** de C++.

- Captura de excepciones:

```
when ListaDeErrores do Instrucción
```

Similar a la cláusula **catch** de la instrucción **try...catch** de C++. Los errores capturados pueden ser excepciones propiamente dichas o errores reportados con la variable SQLCODE. Estos últimos errores se producen al ejecutarse instrucciones SQL. Las instrucciones **when** deben colocarse al final de los procedimientos.

- Instrucciones de control:

```
exit;  
suspend;
```

La instrucción **exit** termina la ejecución del procedimiento actual, y es similar a la instrucción **return** de C++. Por su parte, **suspend** se utiliza en procedimientos que devuelven un conjunto de filas para retornar valores a la rutina que llama a este procedimiento. Con esta última instrucción, se interrumpe temporalmente el procedimiento, hasta que la rutina que lo llama haya procesado los valores retornados.

- Instrucciones compuestas:

```
begin ListaDeInstrucciones end
```

La sintaxis de los procedimientos de InterBase es similar a la de Pascal. A diferencia de este último lenguaje, la palabra **end** no puede tener un punto y coma a continuación.

Mostraré ahora un par de procedimientos sencillos, que ejemplifiquen el uso de estas instrucciones. El siguiente procedimiento, basado en las tablas definidas en el capítulo sobre DDL, sirve para recalcular la suma total de un pedido, si se suministra el número de pedido correspondiente:

```
create procedure RecalcularTotal(NumPed int) as  
declare variable Total integer;  
begin  
  select sum(Cantidad * PVP * (100 - Descuento) / 100)  
  from Detalles, Articulos  
  where Detalles.RefArticulo = Articulos.Codigo  
  and Detalles.RefPedido = :NumPed  
  into :Total;  
  if (Total is null) then  
    Total = 0;
```

```

update Pedidos
set Total = :Total
where Numero = :NumPed;
end ^

```

El procedimiento consiste básicamente en una instrucción **select** que calcula la suma de los totales de todas las líneas de detalles asociadas al pedido; esta instrucción necesita mezclar datos provenientes de las líneas de detalles y de la tabla de artículos. Si el valor total es nulo, se cambia a cero. Esto puede suceder si el pedido no tiene líneas de detalles; en este caso, la instrucción **select** retorna el valor nulo. Finalmente, se localiza el pedido indicado y se le actualiza el valor a la columna *Total*, utilizando el valor depositado en la variable local del mismo nombre.

El procedimiento que definimos a continuación se basa en el anterior, y permite recalcular los totales de todas las filas almacenadas en la tabla de pedidos; de este modo ilustramos el uso de las instrucciones **for select do** y **execute procedure**:

```

create procedure RecalcularPedidos as
declare variable Pedido integer;
begin
for select Numero from Pedidos into :Pedido do
execute procedure RecalcularTotal :Pedido;
end ^

```

Procedimientos que devuelven un conjunto de datos

Antes he mencionado la posibilidad de superar las restricciones de las expresiones **select** del modelo relacional mediante el uso de procedimientos almacenados. Un procedimiento puede diseñarse de modo que devuelva un conjunto de filas; para esto hay que utilizar la instrucción **suspend**, que transfiere el control temporalmente a la rutina que llama al procedimiento, para que ésta pueda hacer algo con los valores asignados a los parámetros de salida. Esta técnica es poco habitual en los lenguajes de programación más extendidos; si quiere encontrar algo parecido, puede desenterrar los *iteradores* del lenguaje CLU, diseñado por Barbara Liskov a mediados de los setenta.

Supongamos que necesitamos obtener la lista de los primeros veinte, treinta o mil cuatrocientos números primos. Comencemos por algo fácil, con la función que analiza un número y dice si es primo o no:

```

create procedure EsPrimo(Numero integer)
returns (Respuesta integer) as
declare variable I integer;
begin
I = 2;
while (I < Numero) do
begin
if (cast((Numero / I) as integer) * I = Numero) then

```

```

begin
    Respuesta = 0;
    exit;
end
I = I + 1;
end
Respuesta = 1;
end ^

```

Ya sé que hay implementaciones más eficientes, pero no quería complicar mucho el ejemplo. Observe, de paso, la pirueta que he tenido que realizar para ver si el número es divisible por el candidato a divisor. He utilizado el criterio del lenguaje C para las expresiones lógicas: devuelvo 1 si el número es primo, y 0 si no lo es. Recuerde que InterBase no tiene un tipo *Boolean*.

Ahora, en base al procedimiento anterior, implementamos el nuevo procedimiento *Primos*:

```

create procedure Primos(Total integer)
    returns (Primo Integer) as
declare variable I integer;
declare variable Respuesta integer;
begin
    I = 0;
    Primo = 2;
    while (I < Total) do
    begin
        execute procedure EsPrimo Primo
            returning_values Respuesta;
        if (Respuesta = 1) then
        begin
            I = I + 1;
            suspend; /* ;;; Nuevo !!! */
        end
        Primo = Primo + 1;
    end
end ^

```

Este procedimiento puede ejecutarse en dos contextos diferentes: como un procedimiento normal, o como procedimiento de selección. Como procedimiento normal, utilizamos la instrucción **execute procedure**, como hasta ahora:

```
execute procedure Primos(100);
```

No obstante, no van a resultar tan sencillas las cosas. Esta llamada, si se realiza desde *Windows ISQL*, solamente devuelve el primer número primo (era el 2, ¿o no?). El problema es que, en ese contexto, la primera llamada a **suspend** termina completamente el algoritmo.

La segunda posibilidad es utilizar el procedimiento *como si fuera una tabla o vista*. Desde *Windows ISQL* podemos lanzar la siguiente instrucción, que nos mostrará los primeros cien números primos:

```
select * from Primos(100);
```

Por supuesto, el ejemplo anterior se refiere a una secuencia aritmética. En la práctica, un procedimiento de selección se implementa casi siempre llamando a **suspend** dentro de una instrucción **for...do**, que recorre las filas de una consulta.

Recorriendo un conjunto de datos

En esta sección mostraré un par de ejemplos más complicados de procedimientos que utilizan la instrucción **for...select** de InterBase. El primero tiene que ver con un sistema de entrada de pedidos. Supongamos que queremos actualizar las existencias en el inventario después de haber grabado un pedido. Tenemos dos posibilidades, en realidad: realizar esta actualización mediante un *trigger* que se dispare cada vez que se guarda una línea de detalles, o ejecutar un procedimiento almacenado al finalizar la grabación de todas las líneas del pedido.

La primera técnica será explicada en breve, pero adelanto en estos momentos que tiene un defecto. Pongamos como ejemplo que dos usuarios diferentes están pasando por el cajero, simultáneamente. El primero saca un *pack* de Coca-Colas de la cesta de la compra, mientras el segundo pone Pepsis sobre el mostrador. Si, como es de esperar, la grabación del pedido tiene lugar mediante una transacción, al dispararse el *trigger* se han modificado las filas de estas dos marcas de bebidas, y se han bloqueado hasta el final de la transacción. Ahora, inesperadamente, el primer usuario saca Pepsis mientras el segundo nos sorprende con Coca-Colas; son unos fanáticos de las bebidas americanas estos individuos. El problema es que el primero tiene que esperar a que el segundo termine para poder modificar la fila de las Pepsis, mientras que el segundo se halla en una situación similar.

Esta situación se denomina *abrazo mortal (deadlock)* y realmente no es problema alguno para InterBase, en el cual los procesos fallan inmediatamente cuando se les niega un bloqueo⁶. Pero puede ser un peligro en otros sistemas con distinta estrategia de espera. La solución más común consiste en que cuando un proceso necesita bloquear ciertos recursos, lo haga siempre en el mismo orden. Si nuestros dos consumidores de líquidos oscuros con burbujas hubieran facturado sus compras en orden alfabético, no se hubiera producido este conflicto. Por supuesto, esto descarta el uso de un *trigger* para actualizar el inventario, pues hay que esperar a que estén todos los

⁶ Realmente, es el BDE quien configura a InterBase de este modo. En la versión 5.0.1.24 se introduce el nuevo parámetro *WAIT ON LOCKS*, para modificar este comportamiento.

productos antes de ordenar y realizar entonces la actualización. El siguiente procedimiento se encarga de implementar el algoritmo explicado:

```

create procedure ActualizarInventario(Pedido integer) as
declare variable CodArt integer;
declare variable Cant integer;
begin
    for select RefArticulo, Cantidad
        from Detalles
        where RefPedido = :Pedido
        order by RefArticulo
        into :CodArt, :Cant do
        update Articulos
        set Pedidos = Pedidos + :Cant
        where Codigo = :CodArt;
end ^

```

Otro ejemplo: necesitamos conocer los diez mejores clientes de nuestra tienda. Pero sólo los diez primeros, y no vale mirar hacia otro lado cuando aparezca el undécimo. Algunos sistemas SQL tienen extensiones con este propósito (**top** en SQL Server; **fetch first** en DB2), pero no InterBase. Este procedimiento, que devuelve un conjunto de datos, nos servirá de ayuda:

```

create procedure MejoresClientes(Rango integer)
returns (Codigo int, Nombre varchar(30), Total int) as
begin
    for select Codigo, Nombre, sum(Total)
        from Clientes, Pedidos
        where Clientes.Codigo = Pedidos.Cliente
        order by 3 desc
        into :Codigo, :Nombre, :Total do
    begin
        suspend;
        Rango = Rango - 1;
        if (Rango = 0) then
            exit;
        end
    end
end ^

```

Entonces podremos realizar consultas como la siguiente:

```

select *
from MejoresClientes(10)

```

Triggers, o disparadores

Una de las posibilidades más interesantes de los sistemas de bases de datos relacionales son los *triggers*, o disparadores; en adelante, utilizaré preferentemente la palabra inglesa original. Se trata de un tipo de procedimiento almacenado que se activa automáticamente al efectuar operaciones de modificación sobre ciertas tablas de la base de datos.

La sintaxis de la declaración de un *trigger* es la siguiente:

```
create trigger NombreTrigger for Tabla [active | inactive]
  {before | after} {delete | insert | update}
  [position Posición]
  as CuerpoDeProcedimiento
```

El cuerpo de procedimiento tiene la misma sintaxis que los cuerpos de los procedimientos almacenados. Las restantes cláusulas del encabezamiento de esta instrucción tienen el siguiente significado:

Cláusula	Significado
<i>NombreTrigger</i>	El nombre que se le va a asignar al <i>trigger</i>
<i>Tabla</i>	El nombre de la tabla a la cual está asociado
active inactive	Puede crearse inactivo, y activarse después
before after	Se activa antes o después de la operación
delete insert update	Qué operación provoca el disparo del <i>trigger</i>
position	Orden de disparo para la misma operación

A diferencia de otros sistemas de bases de datos, los *triggers* de InterBase se definen para una sola operación sobre una sola tabla. Si queremos compartir código para eventos de actualización de una o varias tablas, podemos situar este código en un procedimiento almacenado y llamarlo desde los diferentes *triggers* definidos.

Un parámetro interesante es el especificado por **position**. Para una operación sobre una tabla pueden definirse varios *triggers*. El número indicado en **position** determina el orden en que se disparan los diferentes sucesos; mientras más bajo sea el número, mayor será la prioridad. Si dos *triggers* han sido definidos con la misma prioridad, el orden de disparo entre ellos será aleatorio.

Hay una instrucción similar que permite modificar algunos parámetros de la definición de un *trigger*, como su orden de disparo, si está activo o no, o incluso su propio cuerpo:

```
alter trigger NombreTrigger [active | inactive]
  [{before | after} {delete | insert | update}]
  [position Posición]
  [as CuerpoProcedimiento]
```

Podemos eliminar completamente la definición de un *trigger* de la base de datos mediante la instrucción:

```
drop trigger NombreTrigger
```

Las variables *new* y *old*

Dentro del cuerpo de un *trigger* pueden utilizarse las variables predefinidas *new* y *old*. Estas variables hacen referencia a los valores nuevos y anteriores de las filas involucradas en la operación que dispara el *trigger*. Por ejemplo, en una operación de modificación **update**, *old* se refiere a los valores de la fila antes de la modificación y *new* a los valores después de modificados. Para una inserción, solamente tiene sentido la variable *new*, mientras que para un borrado, solamente tiene sentido *old*.

El siguiente *trigger* hace uso de la variable *new*, para acceder a los valores del nuevo registro después de una inserción:

```

create trigger UltimaFactura for Pedidos
  active after insert position 0 as
declare variable UltimaFecha date;
begin
  select UltimoPedido
  from Clientes
  where Codigo = new.RefCliente
  into :UltimaFecha;
  if (UltimaFecha < new.FechaVenta) then
    update Clientes
    set UltimoPedido = new.FechaVenta
    where Codigo = new.RefCliente;
end ^

```

Este *trigger* sirve de contraejemplo a un error muy frecuente en la programación SQL. La primera instrucción busca una fila particular de la tabla de clientes y, una vez encontrada, extrae el valor de la columna *UltimoPedido* para asignarlo a la variable local *UltimaFecha*. El error consiste en pensar que esta instrucción, a la vez, deja a la fila encontrada como “fila activa”. El lenguaje de *triggers* y procedimientos almacenados de InterBase, y la mayor parte de los restantes sistemas, no utiliza “filas activas”. Es por eso que en la instrucción **update** hay que incluir una cláusula **where** para volver a localizar el registro del cliente. De no incluirse esta cláusula, cambiaríamos la fecha para *todos* los clientes.

Es posible cambiar el valor de una columna correspondiente a la variable *new*, pero solamente si el *trigger* se define “antes” de la operación de modificación. En cualquier caso, el nuevo valor de la columna se hace efectivo después de que la operación tenga lugar.

Más ejemplos de *triggers*

Para mostrar el uso de *triggers*, las variables *new* y *old* y los procedimientos almacenados, mostraré cómo se puede actualizar automáticamente el inventario de artículos

y el total almacenado en la tabla de pedidos en la medida en que se realizan actualizaciones en la tabla que contiene las líneas de detalles.

Necesitaremos un par de procedimientos auxiliares para lograr una implementación más modular. Uno de estos procedimientos, *RecalcularTotal*, debe actualizar el total de venta de un pedido determinado, y ya lo hemos programado antes. Repito aquí su código, para mayor comodidad:

```

create procedure RecalcularTotal(NumPed int) as
declare variable Total integer;
begin
    select sum(Cantidad * PVP * (100 - Descuento) / 100)
    from Detalles, Articulos
    where Detalles.RefArticulo = Articulos.Codigo
    and Detalles.RefPedido = :NumPed
    into :Total;
    if (Total is null) then
        Total = 0;
    update Pedidos
    set Total = :Total
    where Numero = :NumPed;
end ^

```

El otro procedimiento debe modificar el inventario de artículos. Su implementación es muy simple:

```

create procedure ActInventario(CodArt integer, Cant Integer) as
begin
    update Articulos
    set Pedidos = Pedidos + :Cant
    where Codigo = :CodArt;
end ^

```

Ahora le toca el turno a los *triggers*. Los más sencillos son los relacionados con la inserción y borrado; en el primero utilizaremos la variable *new*, y en el segundo, *old*:

```

create trigger NuevoDetalle for Detalles
    active after insert position 1 as
begin
    execute procedure RecalcularTotal new.RefPedido;
    execute procedure ActInventario
        new.RefArticulo, new.Cantidad;
end ^

create trigger EliminarDetalle for Detalles
    active after delete position 1 as
declare variable Decremento integer;
begin
    Decremento = - old.Cantidad;
    execute procedure RecalcularTotal old.RefPedido;
    execute procedure ActInventario
        old.RefArticulo, :Decremento;
end ^

```

Es curiosa la forma en que se pasan los parámetros a los procedimientos almacenados. Tome nota, en particular, de que hemos utilizado una variable local, *Decremento*, en el *trigger* de eliminación. Esto es así porque no se puede pasar expresiones como parámetros a los procedimientos almacenados, ni siquiera para los parámetros de entrada.

Finalmente, nos queda el *trigger* de modificación:

```

create trigger ModificarDetalle for Detalles
  active after update position 1 as
declare variable Decremento integer;
begin
  execute procedure RecalcularTotal new.RefPedido;
  if (new.RefArticulo <> old.RefArticulo) then
  begin
    Decremento = -old.Cantidad;
    execute procedure ActInventario
      old.RefArticulo, :Decremento;
    execute procedure ActInventario
      new.RefArticulo, new.Cantidad;
  end
  else
  begin
    Decremento = new.Cantidad - old.Cantidad;
    execute procedure ActInventario
      new.RefArticulo, :Decremento;
  end
end ^

```

Observe cómo comparamos el valor del código del artículo antes y después de la operación. Si solamente se ha producido un cambio en la cantidad vendida, tenemos que actualizar un solo registro de inventario; en caso contrario, tenemos que actualizar dos registros. No hemos tenido en cuenta la posibilidad de modificar el pedido al cual pertenece la línea de detalles. Suponemos que esta operación no va a permitirse, por carecer de sentido, en las aplicaciones clientes.

Generadores

Los *generadores* (*generators*) son un recurso de InterBase para poder disponer de valores secuenciales, que pueden utilizarse, entre otras cosas, para garantizar la unicidad de las claves artificiales. Un generador se crea, del mismo modo que los procedimientos almacenados y *triggers*, en un fichero *script* de SQL. El siguiente ejemplo muestra cómo crear un generador:

```

create generatorCodigoEmpleado;

```

Un generador define una variable interna persistente, cuyo tipo es un entero de 32 bits. Aunque esta variable se inicializa automáticamente a 0, tenemos una instrucción para cambiar el valor de un generador:

```
set generatorCodigoEmpleado to 1000;
```

Por el contrario, no existe una instrucción específica que nos permita eliminar un generador. Esta operación debemos realizarla directamente en la tabla del sistema que contiene las definiciones y valores de todos los generadores:

```
delete from rdb$generators
where rdb$generator_name = 'CODIGOEMPLEADO'
```

Para utilizar un generador necesitamos la función *gen_id*. Esta función utiliza dos parámetros. El primero es el nombre del generador, y el segundo debe ser la cantidad en la que se incrementa o decrementa la memoria del generador. La función retorna entonces el valor ya actualizado. Utilizaremos el generador anterior para suministrar automáticamente un código de empleado si la instrucción **insert** no lo hace:

```
create trigger NuevoEmpleado for Empleados
active before insert
as
begin
    if (new.Codigo is null) then
        new.Codigo = gen_id(CodigoEmpleado, 1);
end ^
```

Al preguntar primeramente si el código del nuevo empleado es nulo, estamos permitiendo la posibilidad de asignar manualmente un código de empleado durante la inserción.

Los programas escritos en C++ Builder tienen problemas cuando se asigna la clave primaria de una fila dentro de un *trigger* utilizando un generador, pues el registro recién insertado “desaparece” según el punto de vista de la tabla. Este problema se presenta sólo cuando estamos navegando simultáneamente sobre la tabla.

Para no tener que abandonar los generadores, una de las soluciones consiste en crear un procedimiento almacenado que obtenga el próximo valor del generador, y utilizar este valor para asignarlo a la clave primaria en el evento *BeforePost* de la tabla. En el lado del servidor se programaría algo parecido a lo siguiente:

```
create procedure ProximoCodigo returns (Cod integer) as
begin
    Cod = gen_id(CodigoEmpleado);
end ^
```

En la aplicación crearíamos un componente *spProximoCodigo*, de la clase *TStoredProc*, y lo aprovecharíamos de esta forma en uno de los eventos *BeforePost* o *OnNewRecord* de la tabla de clientes:

```
void __fastcall TmodDatos::tbClientesBeforePost(TDataSet *DataSet)
{
    spProximoCodigo->ExecProc();
    tbClientesCodigo->Value =
        spProximoCodigo->ParamByName("COD")->AsInteger;
}
```

De todos modos, si la tabla cumple determinados requisitos, podemos ahorrarnos trabajo en la aplicación y seguir asignando la clave primaria en el *trigger*. Las condiciones necesarias son las siguientes:

- La tabla no debe tener columnas con valores **default**. Así evitamos que el BDE tenga que releer la fila después de su creación.
- Debe existir un índice único, o casi único, sobre alguna columna alternativa a la clave primaria. La columna de este índice se utilizará entonces como criterio de ordenación para la navegación.
- El valor de la clave primaria no nos importa realmente, como sucede con las claves artificiales.

Las tablas de referencia que abundan en toda base de datos son un buen ejemplo de la clase de tablas anterior. Por ejemplo, si necesitamos una tabla para los diversos valores del estado civil, probablemente la definamos de este modo:

```
create table EstadoCivil (
    Codigo          integer not null primary key,
    Descripcion     varchar(15) not null unique,
    EsperanzaVida  integer not null
);

create generator EstadoCivilGen;

set term ^;

create trigger BIEstadoCivil for EstadoCivil
active before insert as
begin
    Codigo = gen_id(EstadoCivilGen, 1);
end ^
```

En C++ Builder asociaremos una tabla o consulta a la tabla anterior, pero ordenaremos las filas por su descripción, y ocultaremos el campo *Codigo*, que será asignado automáticamente en el servidor. Recuerde, en cualquier caso, que los problemas con la asignación de claves primarias en el servidor son realmente problemas de la navegación con el BDE, y nada tienen que ver con InterBase, en sí.

NOTA IMPORTANTE

En cualquier caso, si necesita valores únicos y *consecutivos* en alguna columna de una tabla, no utilice generadores (ni secuencias de Oracle, o identidades de MS SQL Server). El motivo es que los generadores no se bloquean durante las transacciones. Usted pide un valor dentro de una transacción, y le es concedido; todavía no ha terminado su transacción. A continuación, otro usuario pide el siguiente valor, y sus deseos se cumplen. Pero entonces usted aborta la transacción, por el motivo que sea. La consecuencia: se pierde el valor que recibió, y se produce un "hueco" en la secuencia.

Simulando la integridad referencial

Como hemos explicado, mediante los *triggers* y los procedimientos almacenados podemos expresar reglas de consistencia en forma *imperativa*, en contraste con las reglas *declarativas* que se enuncian al crear tablas: claves primarias, alternativas y externas, condiciones de verificación, etc. En general, es preferible utilizar una regla declarativa antes que su equivalente imperativo. Pero sucede que las posibilidades de las reglas declarativas son más limitadas que las posibilidades de las reglas imperativas.

En InterBase 4, por ejemplo, las restricciones de integridad referencial no admiten modificaciones ni borrados en las tablas maestras de una clave externa. Sin embargo, a veces es deseable permitir estas operaciones y propagar los cambios en cascada a las tablas dependientes.

Ilustraré la forma de lograr restricciones de integridad referencial con propagación de cambios mediante el ejemplo de la tabla de pedidos y líneas de detalles. Recordemos la definición de la tabla de pedidos, en el capítulo sobre el Lenguaje de Definición de Datos:

```
create table Pedidos (
    Numero          int not null,
    RefCliente      int not null,
    RefEmpleado     int,
    FechaVenta     date default "Now",
    Total           int default 0,

    primary key (Numero),
    foreign key (RefCliente) references Clientes (Codigo)
);
```

La definición de la tabla de detalles cambia ahora, sustituyéndose la cláusula **foreign key** que hacía referencia a la tabla de pedidos:

```

create table Detalles (
    RefPedido      int not null,
    NumLinea       int not null,
    RefArticulo    int not null,
    Cantidad       int default 1 not null,
    Descuento      int default 0 not null
                    check (Descuento between 0 and 100),

    primary key (RefPedido, NumLinea),
    foreign key (RefArticulo) references Articulos (Codigo),
    /*
     * Antes: foreign key(RefPedido) references Pedidos(Numero)
     */
    check (RefPedido in (select Numero from Pedidos))
);

```

La nueva cláusula **check** verifica en cada inserción y modificación que no se introduzca un número de pedido inexistente. El borrado en cascada se puede lograr de la siguiente manera:

```

create trigger BorrarDetallesEnCascada for Pedidos
active after delete
position 0
as
begin
    delete from Detalles
    where RefPedido = old.Numero;
end ^

```

Un poco más larga es la implementación de actualizaciones en cascada.

```

create trigger ModificarDetallesEnCascada for Pedidos
active after update
position 0
as
begin
    if (old.Numero <> new.Numero) then
        update Detalles
        set RefPedido = new.Numero
        where RefPedido = old.Numero;
    end ^

```

Por supuesto, los *triggers* hubieran sido mucho más complicados si hubiéramos mantenido la restricción **foreign key** en la declaración de la tabla de detalles, en particular, la propagación de modificaciones.

Excepciones

Sin embargo, todavía no contamos con medios para detener una operación SQL; esta operación sería necesaria para simular imperativamente las restricciones a la propagación de cambios en cascada, en la integridad referencial. Lo que nos falta es poder

lanzar excepciones desde un *trigger* o procedimiento almacenado. Las excepciones de InterBase se crean asociando una cadena de mensaje a un identificador:

```
create exception CLIENTE_CON_PEDIDOS
    "No se puede modificar este cliente"
```

Es necesario confirmar la transacción actual para poder utilizar una excepción recién creada. Existen también instrucciones para modificar el mensaje asociado a una excepción (**alter exception**), y para eliminar la definición de una excepción de la base de datos (**drop exception**).

Una excepción se lanza desde un procedimiento almacenado o *trigger* mediante la instrucción **exception**:

```
create trigger CheckDetails for Clientes
    active before delete
    position 0
as
declare variable Numero int;
begin
    select count(*)
    from Pedidos
    where RefCliente = old.Codigo
    into :Numero;
    if (:Numero > 0) then
        exception CLIENTE_CON_PEDIDOS;
    end ^
```

Las excepciones de InterBase determinan que cualquier cambio realizado dentro del cuerpo del *trigger* o procedimiento almacenado, sea directa o indirectamente, se anule automáticamente. De esta forma puede programarse algo parecido a las transacciones anidadas de otros sistemas de bases de datos.

Si la instrucción **exception** es similar a la instrucción **throw** de C++, el equivalente más cercano a **try...catch** es la instrucción **when** de InterBase. Esta instrucción tiene tres formas diferentes. La primera intercepta las excepciones lanzadas con **exception**:

```
when exception NombreExcepción do
    BloqueInstrucciones;
```

Con la segunda variante, se detectan los errores producidos por las instrucciones SQL:

```
when sqlcode Numero do
    BloqueInstrucciones;
```

Los números de error de SQL aparecen documentados en la ayuda en línea y en el manual *Language Reference* de InterBase. A grandes rasgos, la ejecución correcta de una

instrucción devuelve un código igual a 0, cualquier valor negativo es un error propiamente dicho (-803, por ejemplo, es un intento de violación de una clave primaria), y los valores positivos son advertencias. En particular, 100 es el valor que se devuelve cuando una selección única no encuentra el registro buscado. Este convenio es parte del estándar de SQL, aunque los códigos de error concreto varíen de un sistema a otro.

La tercera forma de la instrucción **when** es la siguiente:

```
when gáscode Numero do
    BloqueInstrucciones;
```

En este caso, se están interceptando los mismos errores que con **sqlcode**, pero se utilizan los códigos internos de InterBase, que ofrecen más detalles sobre la causa. Por ejemplo, los valores 335544349 y 35544665 corresponden a -803, la violación de unicidad, pero el primero se produce cuando se inserta un valor duplicado en cualquier índice único, mientras que el segundo se reserva para las violaciones específicas de clave primaria o alternativa.

En cualquier caso, las instrucciones **when** deben ser las últimas del bloque en que se incluyen, y pueden colocarse varias simultáneamente, para atender varios casos:

```
begin
    /* Instrucciones */
    /* ... */
    when sqlcode -803 do
        Resultado = "Violación de unicidad";
    when exception CLIENTE_CON_PEDIDOS do
        Resultado = "Elimine primero los pedidos realizados";
end
```

La Tercera Regla de Martens sigue siendo aplicable a estas instrucciones: no detenga la propagación de una excepción, a no ser que tenga una solución a su causa.

Alertadores de eventos

Los alertadores de eventos (*event alerter*s) son un recurso único, por el momento, de InterBase. Los procedimientos almacenados y *triggers* de InterBase pueden utilizar la instrucción siguiente:

```
post_event NombreDeEvento
```

El nombre de evento puede ser una constante de cadena o una variable del mismo tipo. Cuando se produce un evento, InterBase avisa a todos los clientes interesados de la ocurrencia del mismo.

Los alertadores de eventos son un recurso muy potente. Sitúese en un entorno cliente/servidor donde se producen con frecuencia cambios en una base de datos. Las estaciones de trabajo normalmente no reciben aviso de estos cambios, y los usuarios deben actualizar periódica y frecuentemente sus pantallas para reflejar los cambios realizados por otros usuarios, pues en caso contrario puede suceder que alguien tome una decisión equivocada en base a lo que está viendo en pantalla. Sin embargo, refrescar la pantalla toma tiempo, pues hay que traer cierta cantidad de información desde el servidor de bases de datos, y las estaciones de trabajo realizan esta operación periódicamente, colapsando la red. El personal de la empresa se aburre en los tiempos de espera, la moral se resquebraja y la empresa se sitúa al borde del caos...

Entonces aparece Usted, un experto programador de C++ Builder e InterBase, y añade *triggers* a discreción a la base de datos, en este estilo:

```
create trigger AlertarCambioBolsa for Cotizaciones
    active after update position 10
as
begin
    post_event "CambioCotizacion";
end ^
```

Observe que se ha definido una prioridad baja para el orden de disparo del *trigger*. Hay que aplicar la misma técnica para cada una de las operaciones de actualización de la tabla de cotizaciones.

Luego, en el módulo de datos de la aplicación que se ejecuta en las estaciones de trabajo, hay que añadir el componente *TIBEventAlerter*, que se encuentra en la página *Samples* de la Paleta de Componentes. Este componente tiene las siguientes propiedades, métodos y eventos:

Nombre	Tipo	Propósito
<i>Events</i>	Propiedad	Los nombres de eventos que nos interesan.
<i>Registered</i>	Propiedad	Debe ser <i>True</i> para notificar, en tiempo de diseño, nuestro interés en los eventos almacenados en la propiedad anterior.
<i>Database</i>	Propiedad	La base de datos a la cual nos conectaremos.
<i>RegisterEvents</i>	Método	Notifica a la base de datos nuestro interés por los eventos de la propiedad <i>Events</i> .
<i>UnRegisterEvents</i>	Método	El inverso del método anterior.
<i>OnEvent.Alert</i>	Evento	Se dispara cada vez que se produce el evento.

En nuestro caso, podemos editar la propiedad *Events* y teclear la cadena *CambioCotizacion*, que es el nombre del evento que necesitamos. Conectamos la propiedad *Database* del componente a nuestro componente de bases de datos y activamos la propiedad *Registered*. Luego creamos un manejador para el evento *OnEvent.Alert* similar a éste:

```

void __fastcall TForm1::IBEventAlerter1EventAlert(TObject *Sender,
    AnsiString EventName, long EventCount, bool &CancelAlerts)
{
    tbCotizaciones->Refresh();
}

```

Cada vez que se modifique el contenido de la tabla *Cotizaciones*, el servidor de InterBase lanzará el evento identificado por la cadena *CambioCotizacion*, y este evento será recibido por todas las aplicaciones interesadas. Cada aplicación realizará consecuentemente la actualización visual de la tabla en cuestión.

Esta historia termina previsiblemente. La legión de usuarios del sistema lo aclama con fervor, su jefe le duplica el salario, usted se casa ... o se compra un perro ... o ... Bueno, se me ha complicado un poco el guión; póngale usted su final preferido.

Funciones de usuario en InterBase

Para finalizar el capítulo, mostraré un ejemplo de cómo utilizar las DLL para extender la funcionalidad de un servidor de InterBase. Como forma de ampliar el conjunto de funciones disponibles en SQL, los servidores de InterBase basados en Windows 95 y Windows NT admiten la creación de *funciones definidas por el usuario* (*User Defined Functions*, ó *UDF*). Estas funciones se definen en DLLs que se deben registrar en el servidor, para poder ser ejecutadas desde consultas SQL, *triggers* y procedimientos almacenados.

Los pasos para crear una función de usuario son los siguientes:

- Programe la DLL, exportando las funciones deseadas.
- Copie la DLL resultante al directorio *bin* del servidor de InterBase. Si se trata de un servidor local, o si tenemos acceso al disco duro del servidor remoto, esto puede realizarse cambiando el directorio de salida en las opciones del proyecto.
- Utilice la instrucción **declare external function** de InterBase para registrar la función en la base de datos correspondiente. Para facilitar el uso de la extensión programada, puede acompañar a la DLL con las declaraciones correspondientes almacenadas en un *script* SQL.

Para ilustrar la técnica, crearemos una función que devuelva el nombre del día de la semana de una fecha determinada. La declaración de la función, en la sintaxis de InterBase, será la siguiente:

```

declare external function DiaSemana(DATE)
    returns cstring(15)
    entry_point "DiaSemana"
    module_name "MisUdfs.dll";

```

Aunque podemos comenzar declarando la función, pues InterBase cargará la DLL sólo cuando sea necesario, es preferible comenzar creando la DLL, así que cree un nuevo proyecto DLL, con el nombre *MisUdfs*.

Las funciones de usuario de InterBase deben implementarse con el atributo `__cdecl`. Hay que tener en cuenta que todos los parámetros se pasan por referencia; incluso los valores de retorno de las funciones se pasan por referencia (se devuelve un puntero), si no se especifica la opción **by value** en la declaración de la función. La correspondencia entre tipos de datos de InterBase y de C++ Builder es sencilla: **int** equivale a **int**, **smallint** a **short int**, las cadenas de caracteres se pasan como punteros a caracteres, y así sucesivamente. En particular, las fechas se pasan en un tipo de registro con la siguiente declaración:

```
typedef struct {
    int Days;
    int Frac;
} TIBDate;
```

Days es la cantidad de días transcurridos a partir de una fecha determinada por InterBase, el 17 de noviembre de 1858. *Frac* es la cantidad de diezmilésimas de segundos transcurridas desde las doce de la noche. Con esta información en nuestras manos, es fácil programar la función *DiaSemana*:

```
__declspec(dllexport) char const * __cdecl DiaSemana(TIBDate &fecha)
{
    static char *dias[] = {
        "Miércoles", "Jueves", "Viernes", "Sábado",
        "Domingo", "Lunes", "Martes" };
    return dias[fecha.Days % 7];
}
```

Para saber qué día de la semana corresponde al “día de la creación” de InterBase, tuvimos que realizar un proceso sencillo de prueba y error; parece que para alguien en este mundo los miércoles son importantes.

Una vez compilado el proyecto, asegúrese que la DLL generada está presente en el directorio *bin* del servidor de InterBase. Active entonces la utilidad *WISQL*, conéctese a una base de datos que contenga tablas con fechas, teclee la instrucción **declare external function** que hemos mostrado anteriormente y ejecútela. A continuación, pruebe el resultado, con una consulta como la siguiente:

```
select DiaSemana(SaleDate), SaleDate,
       cast("Now" as date), DiaSemana("Now")
from   Orders
```

Tenga en cuenta que, una vez que el servidor cargue la DLL, ésta quedará en memoria hasta que el servidor se desconecte. De este modo, para sustituir la DLL (para

añadir funciones o corregir errores) debe primero detener al servidor y volver a iniciarlo posteriormente.

Hay que tener cuidado, especialmente en InterBase 5, con las funciones que devuelven cadenas de caracteres generadas por la DLL. El problema es que estas funciones necesitan un *buffer* para devolver la cadena, que debe ser suministrado por la DLL. No se puede utilizar una variable global con este propósito, como en versiones anteriores de InterBase, debido a la nueva arquitectura multihilos. Ahora todas las conexiones de clientes comparten un mismo proceso en el servidor, y si varias de ellas utilizan una misma UDF, están accediendo a la función desde distintos hilos. Si utilizáramos una variable global, podríamos sobrescribir su contenido con mucha facilidad.

Por ejemplo, ésta es la implementación en C++ Builder de una función de usuario para convertir cadenas a minúsculas:

```
__declspec(dllexport) char * __cdecl Lower(char *s)
{
    int len = strlen(s);
    char *res = (char *) SysGetMem(len + 1);
    // SysGetMem asigna memoria en el formato adecuado
    strcpy(res, s);
    CharLowerBuff(res, len);
    return res;
}
```

Si queremos utilizar esta función desde InterBase, debemos declararla mediante la siguiente instrucción:

```
declare external function lower cstring(256)
returns cstring (256) free_it
entry_point "Lower" module_name "MisUdfs.dll"
```

Observe el uso de la nueva palabra reservada **free_it**, para indicar que la función reserva memoria que debe ser liberada por el servidor.

Transacciones

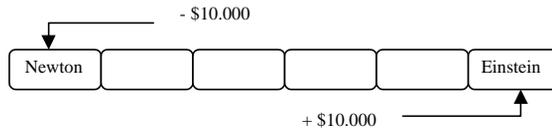
TODO O NADA, Y QUE NO ME ENTERE YO que andas cerca. Parece una amenaza, pero no lo es. La frase anterior puede resumir el comportamiento egoísta deseable para las *transacciones*: el mecanismo que ofrecen las bases de datos para garantizar la coherencia de su contenido, especialmente cuando varias aplicaciones acceden simultáneamente al mismo.

En este capítulo explicamos la teoría general de las transacciones, desde el punto de vista de las bases de datos de escritorio y SQL, y cómo son implementadas por los diferentes sistemas. El programador típico de bases de datos locales asocia la solución de los problemas de concurrencia con la palabra mágica “bloqueos”. Como veremos, esto es sólo parte de la verdad, y en ocasiones, ni siquiera es verdad. Los sistemas profesionales de bases de datos utilizan los bloqueos como un posible mecanismo de implementación del control de concurrencia a bajo nivel. Y el programador debe trabajar y pensar en *transacciones*, como forma de asegurar la consistencia de sus operaciones en la base de datos.

¿Por qué necesitamos transacciones?

Por omisión, cuando realizamos modificaciones en tablas mediante C++ Builder y el BDE, cada operación individual es independiente de las operaciones que le preceden y de las que siguen a continuación. El fallo de una de ellas no afecta a las demás. Sin embargo, existen ocasiones en que nos interesa ligar la suerte de varias operaciones consecutivas sobre una base de datos.

El ejemplo clásico es la transferencia bancaria: hay que restar del saldo de un registro y aumentar en la misma cantidad el saldo de otro. No podemos permitir que, una vez actualizado el primer registro nos encontremos que alguien está trabajando con el segundo registro, y se nos quede el dinero de la transferencia en el limbo. Y no es solución regresar al primer registro y reingresar la cantidad extraída, pues puede que otro usuario haya comenzado a editar este primer registro después de haberlo abandonado nosotros. En este caso nos veríamos como un jugador de béisbol atrapado entre dos bases.

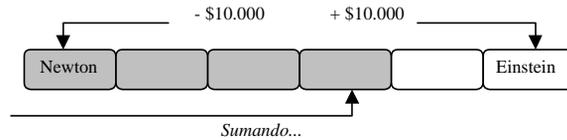


O considere una subida salarial a cierto grupo de empleados de una empresa. En este caso, es mucho más difícil dar marcha atrás a la operación si se produce un error a mediados de la misma. Normalmente, los programadores que vienen del mundo de las bases de datos locales atacan estos problemas blandiendo bloqueos a diestra y siniestra. En el caso de la transferencia, un bloqueo sobre la cuenta destino y la cuenta origen y ¡venga transferencial! En el caso de la subida masiva, un bloqueo sobre la tabla completa, y ¡pobre del que intente acceder a la tabla mientras tanto! Es cierto que los bloqueos son una de las muchas maneras de resolver los problemas de acceso concurrente (aunque no la mejor). Pero una actualización puede fallar por muchos más motivos que por un bloqueo denegado; una violación de alguna restricción de validez puede dejarnos a mitad de una operación larga de actualización sin saber cómo retroceder.

La forma de salir de éste y de otros atolladeros similares es utilizar el concepto de *transacción*. Una transacción es una secuencia de operaciones de lectura y escritura durante las cuales se puede ver la base de datos como un todo consistente y, si se realizan actualizaciones, dejarla en un estado consistente. Estoy consciente de que la oración anterior parece extraída de un libro de filosofía, por lo cual dedicaré los próximos párrafos a despejar la niebla.

En primer lugar: “ver la base de datos como un todo consistente”. Nada que ver con la ecología ni con la Tabla Esmeralda. Con esto quiero decir que, durante todo el intervalo que está activa la transacción, los valores leídos y no modificados por la misma permanecen estables, y si al principio de la misma satisfacían las reglas de integridad, siguen cumpliéndolas durante todo el tiempo de vida de la transacción.

¿Elemental? No tanto. Suponga que una transacción quiere sumar los saldos de las cuentas depositadas en nuestro banco, y comienza a recorrer la tabla pertinente. Suponga también que las filas están almacenadas por orden alfabético de acuerdo al apellido, y que la transacción se encuentra ahora mismo analizando la cuenta de cierto sujeto de apellido Marteens. En ese preciso momento, llega un tal Albert Einstein y quiere transferir diez mil dólares a la cuenta de un tal Isaac Newton (gente importante los clientes de este banco). La fila de Mr. Einstein se actualiza, decrementando su saldo; esta fila ya ha sido leída por la transacción que suma. Luego, la fila de Mr. Newton incrementa su saldo en la cantidad correspondiente. Y esta fila no ha sido leída aún por la transacción sumadora. Por lo tanto, esta transacción al final reportará diez mil dólares de más en el saldo total almacenado; diez mil dólares inexistentes, que es lo más triste del asunto.



En segundo lugar: “si se realizan actualizaciones, dejar la base de datos en un estado consistente”. La condición anterior es necesaria, desde un punto de vista estricto, para el cumplimiento de esta condición, pues no se puede pretender realizar una actualización que satisfaga las reglas de consistencia si la transacción puede partir de un estado no consistente. Pero implica más que esto. En particular, se necesita garantizar que toda la secuencia de operaciones consideradas dentro de una transacción se ejecute; si la transacción aborta a mitad de camino, los cambios efectuados deben poder deshacerse automáticamente. Y esto vale también para el caso especial en que el gato de la chica del piso de abajo entre por la puerta y se electrocute con el cable de alimentación del servidor. Después de retirar a la víctima y reiniciar el servidor, la base de datos no debe acusar recibo de las transacciones inconclusas: el espectáculo debe continuar⁷.

El ácido sabor de las transacciones

A veces, en la literatura anglosajona, se dice que las transacciones tienen propiedades “ácidas”, por las siglas ACID: *Atomicity*, *Consistency*, *Isolation* y *Durability*. O, forzando un poco la traducción para conservar las iniciales: atomicidad, consistencia, independencia y durabilidad. La atomicidad no se refiere al carácter explosivo de las transacciones, sino al hecho de que deben ser *indivisibles*; se realizan todas las operaciones, o no se realiza ninguna. Consistencia quiere decir, precisamente, que una transacción debe llevar la base de datos de un estado consistente a otro. Independencia, porque para ser consistentes debemos imaginar que somos los únicos que tenemos acceso a la base de datos en un instante dado; las demás transacciones deben ser invisibles para nosotros. Y la durabilidad se refiere a que cuando una transacción se confirma, los cambios solamente pueden deshacerse mediante otra transacción.

⁷ Nota del censor: Me estoy dando cuenta de que en este libro se maltrata y abusa de los animales. Antes fue el perro de Codd; ahora, el gato de la vecina... Aunque os parezca mentira, una vez tuve un gato llamado Pink Floyd (aunque sólo respondía por Pinky), y un buen día se le ocurrió morder el cable de alimentación de la tele. Lo curioso es que, a pesar de que se quedó tieso, un oportuno masaje cardíaco lo devolvió a este mundo. Claro, era un gato nuevo y aún no había consumido sus siete vidas.

Transacciones SQL y en bases de datos locales

Para que el sistema de gestión de base de datos reconozca una transacción tenemos que marcar sus límites: cuándo comienza y cuándo termina, además de cómo termina. Todos los sistemas SQL ofrecen las siguientes instrucciones para marcar el principio y el fin de una transacción:

```
start transaction
commit work
rollback work
```

La primera instrucción señala el principio de una transacción, mientras que las dos últimas marcan el fin de la transacción. La instrucción **commit work** señala un final exitoso: los cambios se graban definitivamente; **rollback work** indica la intención del usuario de deshacer todos los cambios realizados desde la llamada a **start transaction**. Solamente puede activarse una transacción por base de datos en cada sesión. Dos usuarios diferentes, sin embargo, pueden tener concurrentemente transacciones activas.

La implementación de transacciones para tablas locales (dBase y Paradox) es responsabilidad del BDE. Las versiones de 16 bits del BDE no ofrecen esta posibilidad. A partir de la versión 3.0 del BDE que apareció con Delphi 2, se soportan las llamadas *transacciones locales*. Esta implementación es bastante limitada, pues no permite deshacer operaciones del lenguaje DDL (**create table**, o **drop table**, por ejemplo), y la independencia entre transacciones es bastante pobre, como veremos más adelante al estudiar los niveles de aislamiento. Tampoco pueden activarse transacciones sobre tablas de Paradox que no tengan definida una clave primaria. Y hay que tener en cuenta la posibilidad de que al cerrar una tabla no se puedan deshacer posteriormente los cambios realizados en la misma, aunque aún no se haya confirmado la transacción.

Otra limitación importante tiene que ver con el hecho de que las transacciones sobre bases de datos de escritorio utilizan bloqueos. Paradox solamente admite hasta 255 bloqueos simultáneos sobre una tabla, y dBase es aun más restrictivo, pues sólo permite 100. Por lo tanto, estos son respectivamente los números máximos de registros que pueden ser modificados en una transacción de Paradox y dBase.

Transacciones implícitas y explícitas

Existen dos formas diferentes en las que una aplicación puede utilizar las transacciones. En la sección anterior he mencionado las instrucciones necesarias para marcar el principio y fin de una transacción, en lenguaje SQL. C++ Builder ofrece métodos en el componente *TDatabase* para ejecutar dichas acciones, que serán estudiados en el

momento adecuado. Sin embargo, lo más frecuente es que el programador no inicie explícitamente transacciones, y que aproveche las transacciones implícitas que puede ofrecer el gestor de bases de datos. En Paradox y dBase esto implica no utilizar transacciones en absoluto.

Los sistemas SQL, en cambio, sí ofrecen transacciones implícitas, que son aprovechadas por el BDE. Si el programador no ha iniciado una transacción desde su programa, cada modificación que la aplicación intente realizar desde la estación de trabajo será englobada en una transacción para esta única operación. ¿La razón? Lo que la aplicación cliente considera una simple actualización, puede significar varias actualizaciones en realidad, si existen *triggers* asociados a la acción de modificación, como vimos en el capítulo anterior. Además, la aplicación puede ejecutar procedimientos almacenados que modifiquen varias filas de la base de datos, y es lógico desear que este conjunto de modificaciones se aplique de forma atómica: todo o nada.

Si está trabajando con InterBase a través del BDE, quizás le interese modificar el parámetro *DRIVER_FLAGS* en el controlador SQL Link. Si asigna 4096 a este parámetro, al confirmar y reiniciar una transacción, se aprovecha el "contexto de transacción" existente, con lo cual se acelera la operación. Estos parámetros serán estudiados en el capítulo sobre el Motor de Datos de Borland.

Ahora bien, cada servidor implementa estas transacciones implícitas de forma diferente, y es aconsejable que probemos cada tipo de servidor antes de confiar la integridad de nuestros datos a este comportamiento por omisión.

La prueba que realizaremos es muy sencilla. Yo utilizaré una base de datos de InterBase que viene con los ejemplos de C++ Builder y Delphi; usted puede usar cualquier otra base de datos SQL equivalente. Mi base de datos está almacenada en el siguiente fichero:

C:\Archivos de programa\Archivos comunes\Borland Shared\Data\MastSql.GDB

Primero nos conectamos a la base de datos con el programa con el programa *InterBase Windows ISQL*, del grupo de programas de InterBase, y creamos el siguiente procedimiento almacenado:

```

create procedure Transferencia(Ordenante int, Beneficiario int,
    Cantidad double precision) as
begin
    update Employee
    set Salary = Salary + :Cantidad
    where EmpNo = :Beneficiario;
    update Employee
    set Salary = Salary - :Cantidad
    where EmpNo = :Ordenante;
end ^

```

Evidentemente, no se trata de una transferencia bancaria “real” (le estamos bajando el salario a un empleado para subírselo a otro) pero nos bastará para lo que pretendemos demostrar. Ahora modifiquemos las reglas sobre la tabla de empleados, de modo que un empleado no pueda tener un salario negativo (¡ya quisieran algunos jefes!):

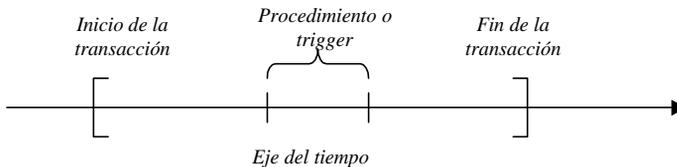
```
alter table Employee
add constraint SalarioPositivo
check (Salary > 0);
```

Intente ahora una “transferencia de salario” que deje con salario negativo al primer empleado. Por ejemplo, puede ejecutar la siguiente instrucción desde el propio *Windows ISQL*, si la base de datos de ejemplos no ha sido aún modificada:

```
execute procedure Transferencia(2, 4, 100000)
```

Naturalmente, como el empleado número 2 no tiene ese salario, la segunda instrucción **update** del procedimiento falla, al no cumplirse la restricción sobre el salario positivo. Pero lo importante es que si listamos los empleados, ¡nos encontraremos que el registro del empleado número 4 no ha sido modificado, a pesar de que aparentemente la primera instrucción se pudo ejecutar exitosamente!

Tome nota de la siguiente característica de InterBase: cada ejecución de un procedimiento o de un trigger inicia una mini-transacción. Si ya hay alguna transacción explícita se trata del único caso en que InterBase permite alguna forma de *transacción anidada*. Si durante la ejecución del procedimiento o trigger se produce una excepción, todas las acciones realizadas en la mini-transacción se anulan. Observe, sin embargo que si ya hay una transacción iniciada, no se deshacen automáticamente las acciones previas a la ejecución del procedimiento. El siguiente diagrama puede ayudarnos a aclarar la idea:



¿Y el resto de los servidores? He realizado la prueba con Oracle 8 y con Microsoft SQL Server 7. En el CD-ROM que acompaña al libro se incluyen *scripts* sencillos para que el lector repita el experimento, si lo desea. Resulta que Oracle se comporta de forma similar a InterBase, ¡pero SQL Server no! En este sistema, la primera instrucción se ejecuta sin problemas, pero cuando la segunda falla, las modificaciones realizadas por la primera permanecen en la base de datos.

Hechos de tal calibre merecen que extraigamos una moraleja. Aunque en teoría C++ Builder trata de forma similar a todos los servidores de datos, en la práctica encontramos diferencias como la que acabamos de presentar, que no hace imposible la programación independiente del formato de datos, pero que nos obliga a prestar suma atención a estos detalles.

Niveles de aislamiento de transacciones

Las propiedades atómicas de las transacciones valen tanto para sistemas multipuesto como para sistemas con un solo usuario. Cuando es posible el acceso simultáneo a una base de datos por varios usuarios o aplicaciones (que pueden residir en la misma máquina), hay que tener en cuenta la forma en que estas transacciones se comportan colectivamente. Para poder garantizar la consistencia de la base de datos cuando varias transacciones se ejecutan concurrentemente sobre la misma, deben cumplirse las siguientes condiciones:

- Una transacción no debe poder leer datos grabados por otra transacción mientras ésta no haya finalizado.
- Los datos leídos por una transacción deben mantenerse constantes hasta que la transacción que los ha leído finalice.

La primera condición es evidente: no podemos tomar una decisión en base a un dato que ha sido colocado tentativamente, que no sabemos aún si viola o no las reglas de consistencia de la base de datos. Solamente las transacciones transforman la base de datos de un estado consistente a otro; pero sólo aquellas transacciones que terminan exitosamente. ¿Elemental, no? Pues ni Paradox ni dBase aíslan a una transacción de cambios no confirmados efectuados desde otros puestos. Comprenderá que esto deja mucho margen para el desastre...

En cuanto al segundo requerimiento, ya hemos hablado acerca de él cuando contábamos la historia de la transferencia bancaria de Albert Einstein. La condición que estamos imponiendo se denomina frecuentemente *lecturas repetibles*, y está motivada por la necesidad de partir de un estado consistente para poder llegar sensatamente a otro estado similar. La violación de esta condición da lugar a situaciones en que la ejecución consecutiva de dos transacciones da un resultado diferente a la ejecución concurrente de las mismas. A esto se le llama, en la jerga académica, el criterio de *serializabilidad* (mi procesador de textos protesta por la palabra, pero yo sé Informática y él no).

En realidad, el criterio de serializabilidad implica también que se prohíba la aparición de *filas fantasmas*: un registro insertado por otra transacción no puede apare-

cer en el campo de visión de una transacción ya iniciada. Pero la mayoría de los sistemas con los que trataremos logran la serializabilidad a la par que las lecturas repetibles.

Considere una aplicación que lee un registro de una tabla para reservar asientos en un vuelo. Esta aplicación se ejecuta concurrentemente en dos puestos diferentes. Llego a uno de los puestos y facturo mi equipaje; frente al otro terminal se sitúa (¡oh, sorpresa!) Pamela Anderson. Antes de comenzar nuestras respectivas transacciones, el registro que almacena la última plaza disponible del vuelo a Tahití contiene el valor 1 (hemos madrugado). Mi transacción lee este valor en la memoria de mi ordenador. Pam hace lo mismo en el suyo. Me quedo atontado mirando a la chica, por lo cual ella graba un 2 en el registro, termina la transacción y se marcha. Regreso a la triste realidad y pulso la tecla para terminar mi transacción. Como había leído un 1 de la base de datos y no hay transacciones en este momento que estén bloqueando el registro, grabo un *dos*, suspiro y me voy. Al montar en el avión descubro que Pam y yo viajamos en el mismo asiento, y uno de los dos tiene que ir sobre las piernas del otro. Esta es una situación embarazosa; para Pamela, claro está.

¿Demasiada verbosa la explicación? Intentémoslo otra vez, pero con menos poesía. Hay una aplicación que ejecuta el siguiente algoritmo:

```
Leer Valor
Valor := Valor + 1
Escribir Valor
```

Está claro que si ejecutamos este algoritmo dos veces, una a continuación de la otra, como resultado, la variable *Valor* debe incrementarse en dos. Pero cuando la ejecución no es secuencial, sino concurrente, las instrucciones elementales que componen el algoritmo pueden entremezclarse. Una de las muchas formas en que puede ocurrir esta mezcla es la siguiente:

<pre>Leer Valor (1) (...espera...) Valor := Valor + 1 (2) (...espera...) (...espera...) Escribir Valor (2)</pre>	<pre>--- Leer Valor (1) (...espera...) Valor := Valor + 1 (2) Escribir Valor (2) ---</pre>
--	--

El resultado final almacenado en *Valor* sería 2, en vez del correcto 3. El problema se presenta cuando la segunda transacción escribe el valor. La primera sigue asumiendo que este valor es 1, pues fue lo que leyó al principio. Si en este momento, el algoritmo relejera la variable, se encontraría con que su contenido ha cambiado. Piense en la vida real: usted saca su billetera, cuenta sus riquezas y se entera de que tiene dos billetes de 100 euros. Un minuto más tarde va a pagar en una tienda, saca la billetera y descubre que uno de los billetes se ha evaporado en la zona crepuscular.

En conclusión, todo sistema de bases de datos debería implementar las transacciones de forma tal que se cumplan las dos condiciones antes expuestas. Pero una implementación tal es algo costosa, y en algunas situaciones se puede prescindir de alguna de las dos condiciones, sobre todo la segunda (y no lo digo por la señorita Anderson). En el estándar del 92 de SQL se definen tres niveles de aislamiento de transacciones, en dependencia de si se cumplen las dos condiciones, si no se cumplen las lecturas repetibles, o si no se cumple ninguna de las condiciones.

C++ Builder tiene previstos estos tres niveles de aislamiento de transacciones, que se configuran en la propiedad *TransIsolation* de los objetos de tipo *TDatabase*. Los valores posibles para esta propiedad son:

Constante	Nivel de aislamiento
<i>tiDirtyRead</i>	Lee cambios sin confirmar
<i>tiReadCommitted</i>	Lee solamente cambios confirmados
<i>tiRepeatableRead</i>	Los valores leídos no cambian durante la transacción

El valor por omisión de *TransIsolation* es *tiReadCommitted*. Aunque el valor almacenado en esta propiedad indique determinado nivel de aislamiento, es prerrogativa del sistema de bases de datos subyacente el aceptar ese nivel o forzar un nivel de aislamiento superior. Por ejemplo, no es posible (ni necesario o conveniente) utilizar el nivel *tiDirtyRead* sobre bases de datos de InterBase. Si una de estas bases de datos se configura para el nivel *tiDirtyRead*, InterBase establece la conexión mediante el nivel *tiReadCommitted*. Por otra parte, como ya hemos mencionado, la implementación actual de las transacciones locales sobre tablas Paradox y dBase solamente admite el nivel de aislamiento *tiDirtyRead*; cualquier otro nivel es aceptado, pero si intentamos iniciar una transacción sobre la base de datos, se nos comunicará el problema.

Registros de transacciones y bloqueos

¿Cómo logran los sistemas de gestión de bases de datos que dos transacciones concurrentes no se estorben entre sí? La mayor parte de los sistemas, cuya arquitectura está basada en el arquetípico System R, utilizan técnicas basadas en bloqueos. Más adelante, al estudiar cómo se actualizan registros con C++ Builder, descubriremos que Paradox y dBase utilizan también bloqueos para garantizar el acceso exclusivo a registros, implementando un control de concurrencia pesimista. Sin embargo, aunque también se trata de “bloqueos”, el significado de los mismos es bastante diferente al que tienen en los sistemas SQL. Ahora veremos solamente cómo se adapta este mecanismo de sincronización a la implementación de transacciones.

En primer lugar, ¿cómo evitar las “lecturas sucias”? Existen dos técnicas básicas. La más sencilla consiste en “marcar” el registro que modifica una transacción como “sucio”, hasta que la transacción confirme o anule sus grabaciones. A esta marca es a

lo que se le llama “bloqueo”. Si otra transacción intenta leer el valor del registro, se le hace esperar hasta que desaparezca la marca sobre el registro. Por supuesto, esta política se basa en un comportamiento “decente” por parte de las transacciones que modifican registros: no deben dejarse cambios sin confirmar o anular por períodos de tiempo prolongados.

Ahora bien, ¿cómo es que la transacción que realiza el cambio restaura el valor original del registro si se decide su anulación? Lo más frecuente, en los sistemas inspirados por System R, es encontrar implementaciones basadas en *registros de transacciones* (*transaction logs*⁸). Estos registros de transacciones son ficheros en los cuales se graban secuencialmente las operaciones necesarias para deshacer las transacciones no terminadas. Algunos guardan en el *log* el valor original; otros, por el contrario, guardan el nuevo valor no confirmado, y lo transfieren a la base de datos solamente al confirmarse la transacción. En la teoría de bases de datos, hablaríamos de *undo* y *redo logs* (*registros para deshacer o rehacer*). Cada técnica tiene sus ventajas y desventajas: por ejemplo, si se utiliza un registro de rehacer y alguien corta la corriente, la base de datos no queda afectada, y al volver a encender el sistema podemos seguir trabajando sobre un estado consistente. Sin embargo, la aplicación de los cambios durante la confirmación es sumamente peligrosa, y el implementador debe tomar precauciones extraordinarias.

Esto nos da una idea para otra política de acceso: si una aplicación, al intentar leer un registro, encuentra que ha sido modificado, puede ir a buscar el valor original. Si utilizamos un *redo log*, el valor es el que se encuentra dentro del propio registro. En caso contrario, hay que buscarlo en el registro de transacciones. Por supuesto, esta técnica es superior a la anterior, pues ofrece mejores tiempos de acceso incluso en caso de modificaciones frecuentes.

Paradox y dBase implementan un *undo log*, pero cuando una aplicación lee un registro modificado por otra transacción, no se toma la molestia de buscar el valor original en el *log file*. Por este motivo es que aparecen lecturas sucias.

Todos los bloqueos impuestos por una transacción, por supuesto, son liberados al terminar ésta, ya sea confirmando o anulando.

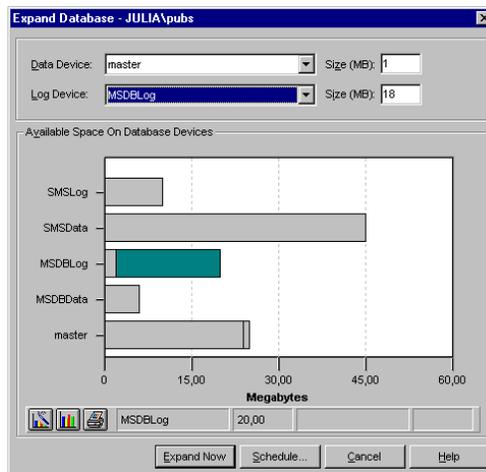
El fichero con el registro de transacciones es una estructura difícil de mantener. Muchos sistemas lo utilizan para la recuperación de bases de datos estropeadas por fallos

⁸ La palabra inglesa *log* quiere decir literalmente “leño”, pero en este contexto se refiere al “cuaderno de bitácora”, en que los capitanes de navío anotaban las incidencias del viaje. En la gloriosa época de la piratería, una de las principales anotaciones era la velocidad de la nave. Para medirla, se arrojaba un *leño* al agua por la proa, y se calculaba el tiempo que tardaba en llegar a la popa. De no haber un madero a mano, se arrojaba al cocinero, con el inconveniente de que el pateo de éste podía distorsionar los resultados.

físicos. Suponga que la última copia de seguridad de su base de datos se realizó el domingo, a medianoche. El lunes por la tarde ocurre lo impensable: encima de su oficina van a montar un bingo, inician las obras de remodelado y las vibraciones de la maquinaria estropean físicamente su disco duro. ¿No se lo cree? Pues a mí me pasó. Supongamos que hemos tenido suerte, y solamente se ha perdido el fichero que contiene la base de datos, pero que el fichero con el *log* está intacto. Entonces, a partir de la última copia y de las transacciones registradas en el fichero, pueden restablecerse los cambios ocurridos durante la jornada del lunes.

Para no tentar a la suerte, los cursos para administradores de bases de datos recomiendan que los ficheros de datos y de transacciones estén situados en discos físicos diferentes. No sólo aumenta la seguridad del sistema, sino que además mejora su eficiencia. Medite en que cada modificación tiene que escribir en ambas estructuras, y que si hay dos controladores físicos para atender a las operaciones de grabación, mejor que mejor.

Por supuesto, el tamaño del registro de transacciones es variable, y difícil de predecir. ¿Y sabe qué? Pues que en SQL Server 6.5 el tamaño del *log file* es estático, por lo que necesita la intervención de un administrador de sistemas para hacerlo crecer cuando está llegando al máximo de su capacidad. La figura siguiente muestra el diálogo que aumenta el tamaño máximo reservado para el registro de transacciones de MS SQL Server 6.5:



Este es un motivo importante para, si ya tiene un SQL Server, se actualice a la versión 7, en la que han resuelto este problema.

Lecturas repetibles mediante bloqueos

Las técnicas expuestas en la sección anterior no garantizan la repetibilidad de las lecturas dentro de una transacción. ¿Recuerda el ejemplo en que dos transacciones incrementaban una misma variable? Estas eran las instrucciones finales de la secuencia conflictiva:

(...espera...)	Escribir Valor (2)
Escribir Valor (2)	---

El problema es que la segunda transacción (la primera en escribir) libera el bloqueo en cuanto termina, lo que sucede antes de que la primera transacción intente su grabación. Eso ... o que ambas transacciones intentan bloquear demasiado tarde. En realidad, hay una solución draconiana a nuestro alcance: todas las transacciones deben anticipar qué registros van a necesitar, para bloquearlos directamente al iniciarse. Claro está, los bloqueos también se liberan en masa al terminar la transacción. A esta estrategia se le denomina *bloqueo en dos fases (two-phase locking)* en los libros de texto, pero no es recomendable en la práctica. En primer lugar, la previsión de las modificaciones no es siempre posible mediante algoritmos. Y en segundo lugar, aún en los casos en que es posible, limita demasiado las posibilidades de concurrencia de las transacciones.

Para una solución intermedia necesitaremos, al menos, dos tipos de bloqueos diferentes: de *lectura* y de *escritura (read locks/write locks)*. La siguiente tabla aparece en casi todos los libros de teoría de bases de datos, y muestra la compatibilidad entre estos tipos de bloqueos:

	Lectura	Escritura
Lectura	Concedido	Denegado
Escritura	Denegado	Denegado

En la nueva estrategia, cuando una transacción va a leer un registro, coloca primeramente un bloqueo de lectura sobre el mismo. De acuerdo a la tabla anterior, la única posibilidad de que este bloqueo le sea denegado es que el registro esté bloqueado en modo de escritura por otra transacción. Y esto es necesario que sea así para evitar las lecturas sucias. Una vez concedido el bloqueo, las restantes transacciones activas pueden leer los valores de este registro, pues se les pueden conceder otros bloqueos de lectura sobre el mismo. Pero no pueden modificar el valor leído, pues lo impide el bloqueo impuesto por la primera transacción. De este modo se garantizan las lecturas repetibles.

Quizás sea necesaria una aclaración: la contención por bloqueos se aplica entre transacciones diferentes. Una transacción puede colocar un bloqueo de lectura sobre un registro y, si necesita posteriormente escribir sobre el mismo, promover el bloqueo a

uno de escritura sin problema alguno. Del mismo modo, un bloqueo de escritura impuesto por una transacción no le impide a la misma realizar otra escritura sobre el registro más adelante.

Por supuesto, todo este mecanismo se complica en la práctica, pues hay que tener en cuenta la existencia de distintos niveles de bloqueos: a nivel de registro, a nivel de página y a nivel de tabla. Estos niveles de bloqueos están motivados por la necesidad de mantener dentro de un tamaño razonable la tabla de bloqueos concedidos por el sistema. Si tenemos un número elevado de bloqueos, el tiempo de concesión o negación de un nuevo bloqueo estará determinado por el tiempo de búsqueda dentro de esta tabla. ¿Recuerda el ejemplo de la transferencia bancaria de Albert Einstein versus la suma de los saldos? La aplicación que suma los saldos de las cuentas debe imponer bloqueos de lectura a cada uno de los registros que va leyendo. Cuando la transacción termine habrá pedido tantos bloqueos como registros tiene la tabla, y esta cantidad puede ser respetable. En la práctica, cuando el sistema detecta que una transacción posee cierta cantidad de bloqueos sobre una misma tabla, trata de promover de nivel a los bloqueos, transformando la multitud de bloqueos de registros en un único bloqueo a nivel de tabla.

Sin embargo, esto puede afectar la capacidad del sistema para hacer frente a múltiples transacciones concurrentes. Tal como hemos explicado en el ejemplo anterior, la transacción del físico alemán debe fallar, pues el registro de su cuenta bancaria ya ha sido bloqueado por la transacción que suma. No obstante, una transferencia entre Isaac Newton y Erwin Schrödinger debe realizarse sin problemas, pues cuando la transacción sumadora va por el registro de Mr. Marteens, los otros dos registros están libres de restricciones. Si, por el contrario, hubiéramos comenzado pidiendo un bloqueo de lectura a nivel de tabla, esta última transacción habría sido rechazada por el sistema.

Se pueden implementar también otros tipos de bloqueo además de los clásicos de lectura y escritura. En particular, las políticas de bloqueo sobre índices representados mediante árboles balanceados son bastante complejas, si se intenta maximizar el acceso concurrente a los datos.

El esquema presentado coincide a grandes rasgos con la forma en que trabaja SQL Server. La implementación de las lecturas repetibles por el SQL Link de Oracle es algo diferente. Oracle genera una versión sólo lectura de los datos leídos en la transacción, y no permite actualizaciones a la aplicación que solicita este nivel de aislamiento.

Variaciones sobre el tema de bloqueos

¿Qué sucede cuando una transacción pide un bloqueo sobre un registro y encuentra que está ocupado temporalmente? Si consultamos los libros clásicos sobre teoría de bases de datos, veremos que casi todos asumen que la aplicación debe esperar a que el sistema pueda concederle el bloqueo. El hecho es, sin embargo, que la mayoría de estos autores se formaron y medraron en el oscuro período en que los ordenadores se programaban mediante tarjetas perforadas, preferiblemente a mano. Para ejecutar una aplicación, había que rellenar montones de formularios para el Administrador del Centro de Cálculo, y lo más probable es que el dichoso programa se ejecutara mientras nosotros no estábamos presentes. Por lo tanto, no se podía contar con la intervención del usuario para resolver el conflicto de intereses entre aplicaciones que luchaban por un mismo registro.

La otra solución extrema al problema del conflicto al bloquear es devolver inmediatamente un error a la aplicación cliente. Aunque parezca mentira a primera vista, ésta es la solución más flexible, pues el programador puede decidir si la aplicación debe esperar y reintentar, o si debe anular inmediatamente la transacción y dedicarse a otra cosa. Y lo más sensato que puede hacer, en esta época de la Informática Interactiva, es pasarle la patata caliente al usuario para que decida.

¿Cómo se comporta cada sistema específico de los que vamos a tratar? Depende, en muchos casos, de cómo está configurado el Motor de Datos de Borland, cuándo éste actúa como capa intermedia. Oracle espera indefinidamente a que el bloqueo conflictivo se libere. SQL Server e Informix permiten configurar en el BDE el tiempo que debe esperar una transacción antes de dar por fallido el intento de bloqueo. En cuanto a InterBase, si la versión del SQL Link es anterior a la 5.0.1.23, se genera una excepción inmediatamente al encontrar un registro bloqueado (no se alarme, espere a leer el resto del capítulo). En realidad, el API de bajo nivel de InterBase puede también soportar directamente el modo de espera, y esto se ha implementado en las versiones más recientes del SQL Link.

El parámetro *WAIT ON LOCKS* del controlador de InterBase es el que determina el comportamiento de InterBase frente a un bloqueo. El tiempo de espera por bloqueos de MS SQL Server se ajusta en el parámetro *TIMEOUT* de su controlador.

Otro problema que se presenta en relación con los bloqueos es conocido como *abrazo mortal*, o *deadlock*. Hay dos niños en una guardería, y una moto y una escopeta de juguete. El primero balbucea: “yo quiero la moto”; y el segundo: “yo quiero la escopeta”. Pero si el objetivo de estos precoces chavales es ir de *easy riders*, lo van a tener complicado, pues no van a obtener la otra mitad del atuendo. Uno de ellos tendrá que renunciar, por lo que será necesaria la actuación de la domadora de la

guardería. Eso, por supuesto, si nuestros niños son de los que esperan indefinidamente a que se libere el “bloqueo” sobre el juguete deseado. Si son de los que se rinden a la primera (¡es lo preferible en este caso!), en cuanto el primero solicita la escopeta y ve que está en uso, cambia de juego y deja al segundo en libertad para ir por las carreteras violando la ley.

Existe una sencilla receta para minimizar la aparición de abrazos mortales, y que es de particular importancia en aquellos sistemas que hacen esperar a las aplicaciones por los bloqueos que solicitan:

“Obligue a las transacciones a bloquear siempre en el mismo orden”

Por ejemplo, “moto” y “escopeta”. Enséñeles el alfabeto a los niños, para que cuando pidan varios juguetes, lo hagan en orden alfabético (¿no es mucho pedir para un *easy rider*?). Otro ejemplo, extraído del así llamado mundo real: en un sistema de entrada de pedidos hay que actualizar las existencias en almacén de los artículos contenidos en un pedido. Entonces, lo más sensato es ordenar las actualizaciones de acuerdo al código o al nombre del artículo. Así se evitan abrazos mortales entre pedidos que venden C++ Builder/Delphi, y Delphi/C++ Builder.

El jardín de los senderos que se bifurcan

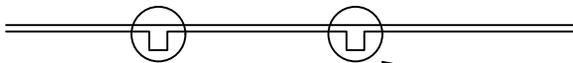
InterBase resuelve los problemas de concurrencia con una técnica diferente a las utilizadas por los demás sistemas de bases de datos. Si le atrae la ciencia ficción, los viajes en el tiempo y la teoría de los mundos paralelos, le gustará también la siguiente explicación.

Volvamos al ejemplo de las maniobras financieras de la comunidad internacional de físicos, suponiendo esta vez que el sistema de base de datos empleado es InterBase. Este día, Mr. Marteens, que no es físico ni matemático, sino banquero (por lo tanto, un hombre feliz), llega a su banco temprano en la mañana. Para Marteens levantarse a las nueve de la mañana es madrugar, y es a esa hora que inicia una transacción para conocer cuán rico es. Recuerde esta hora: las nueve de la mañana.

A las nueve y diez minutos se presenta Albert Einstein en una de las ventanas de la entidad a mover los famosos diez mil dólares de su cuenta a la cuenta de Newton. Si Ian Marteens no hubiese sido programador en una vida anterior y hubiera escogido para el sistema informático de su banco un sistema de gestión implementado con bloqueos, Einstein no podría efectuar su operación hasta las 9:30, la hora en que profetizamos que terminará la aplicación del banquero. Sin embargo, esta vez Albert logra efectuar su primera operación: extraer el dinero de su cuenta personal. Nos lo imaginamos pasándose la mano por la melena, en gesto de asombro: ¿qué ha sucedido?

Bueno, mi querido físico, ha sucedido que el Universo de datos almacenados se ha dividido en dos mundos diferentes: el mundo de Marteens, que corresponde a los datos existentes a las nueve de la mañana, y el mundo de Einstein, que acusa todavía un faltante de \$10.000, pues la transacción no ha terminado. Para que esto pueda ocurrir, deben existir dos versiones diferentes de la cuenta bancaria de Einstein, una en cada Universo. La versión del físico es todavía una versión tentativa; si la señora Einstein introduce la tarjeta en un cajero automático para averiguar el saldo de la cuenta de su marido, no tendrá acceso a la nueva versión, y no se enterará de las locuras inversionistas de su cónyuge. En este punto insistiremos más adelante.

El Universo, según Marteens



Dos versiones del mismo registro

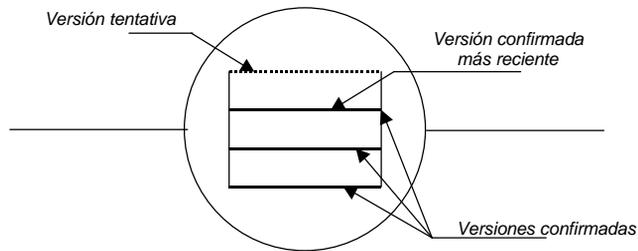
El Universo, según Einstein

Algo parecido sucederá cuando Einstein modifique la cuenta de Newton, incrementándola en la cantidad extraída. Esta vez también se creará una nueva versión que no será visible para la transacción de las 9:00. *Ni siquiera cuando Einstein confirme la transacción.* La idea es que cada transacción solamente ve el mundo tal como era en el momento en que se inicia. Es como si cada transacción sacara una copia local de los datos que va a utilizar. De hecho, hay algunos sistemas que utilizan técnicas parecidas de *replicación*, para garantizar las lecturas repetibles. InterBase *no* hace esto. InterBase saca copias de la parte de la base de datos afectada por actualizaciones concurrentes; de esta manera, se mantiene la base de datos dentro de un tamaño razonable.

Los problemas de este enfoque se producen cuando los Universos deben volver a sincronizarse. Por ejemplo, ¿qué sucede cuando Einstein confirma su transacción? Nada: siguen existiendo dos versiones de su cuenta. La más reciente es la modificada, y si alguna transacción comienza después de que esta confirmación ocurra, la versión que verá es la grabada por Einstein. La versión de las 9:00 existe solamente porque hay una transacción que la necesita hasta las 9:30; a partir de ese momento, pierde su razón de ser y desaparece.

Pero no siempre las cosas son tan fáciles. Mientras Einstein realizaba su transferencia, Newton, que hacía las compras en el supermercado (hay algunos que abren muy temprano), intentaba pagar con su tarjeta. Iniciaba una transacción, durante la cual extraía dinero de su cuenta; Newton no puede ver, por supuesto, los cambios realizados por Einstein, al no estar confirmada la transacción. En este caso, Newton no puede modificar el registro de su cuenta, pues InterBase solamente permite una versión sin confirmar por cada registro.

Es útil, por lo tanto, distinguir entre versiones “tentativas”, que pertenecen a transacciones sin confirmar, y versiones “definitivas”, que pertenecen a transacciones ya confirmadas:



Solamente puede haber una versión tentativa por registro. Esta restricción actúa, desde el punto de vista de las aplicaciones clientes, exactamente igual que un bloqueo de escritura a nivel de registro. Cuando InterBase detecta que alguien intenta crear una versión de un registro que ya tiene una versión tentativa, lanza el siguiente mensaje de error; extraño y confuso, según mi humilde opinión:

“A deadlock was detected”

Sin embargo, sí se permiten varias versiones confirmadas de un mismo registro. Si una aplicación modifica un registro, y hay otra transacción activa que ha leído el mismo registro, la versión antigua se conserva hasta que la transacción desfasada culmina su ciclo de vida. Evidentemente, se pueden acumular versiones obsoletas, pero en cualquier caso, cuando se conecta una nueva transacción, siempre recibe la versión confirmada más reciente.

Con las versiones de registros de InterBase sucede igual que con las creencias humanas: no importa si se ha demostrado que tal o más cual creencia es falsa. Siempre que alguna pobre alma tenga fe, el objeto en que se cree “existe”. Y si duda de mi palabra, pregúntele al Sr. Berkeley.

¿Bloqueos o versiones?

En comparación con las técnicas de aislamiento basadas en la contención (bloqueos), la técnica empleada por InterBase, conocida como *Arquitectura Multigeneracional*, se puede calificar de optimista. Cuando se trata de optimismo y pesimismo en relación con la implementación del aislamiento entre transacciones, las ventajas de una estrategia optimista son realmente abrumadoras. Los sistemas basados en bloqueos han sido diseñados y optimizados con la mente puesta en un tipo de transacciones conocidas como OLTP, de las siglas inglesas *OnLine Transaction Processing* (procesamiento

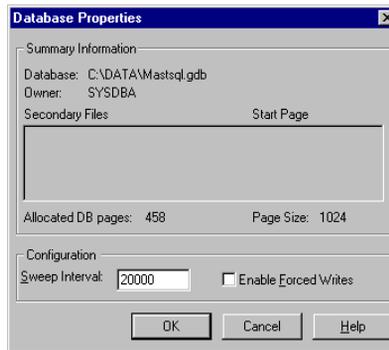
de transacciones en línea). Estas transacciones se caracterizan por su breve duración, y por realizar preferentemente escrituras. Como las transacciones proceden por ráfagas, y cada una involucra a unos pocos registros, la posibilidad de un conflicto entre un par de ellas es pequeña. Como ejemplo práctico, piense en cómo funciona una base de datos que alimenta a una red de cajeros automáticos. Evidentemente, la técnica de bloqueos a nivel de registro funciona estupendamente bajo estas suposiciones. Y lo que también es de notar, la técnica optimista también da la talla en este caso.

Sin embargo, existen otros tipos de transacciones que estropean la fiesta. Son las utilizadas por los sistemas denominados *DSS: Decision Support Systems*, o sistemas de ayuda para las decisiones. El ejemplo de la transacción que suma los saldos, y que hemos utilizado a lo largo del capítulo, es un claro ejemplar de esta especie. También las aplicaciones que presentan gráficos, estadísticas, que imprimen largos informes... Estas transacciones se caracterizan por un tiempo de vida relativamente prolongado, y por preferir las operaciones de lectura.

¿Otro ejemplo importante?, el proceso que realiza la copia de seguridad de la base de datos. ¡La transacción iniciada por este proceso debe tener garantizadas las lecturas repetibles, o podemos quedarnos con una copia inconsistente de la base de datos!

En un sistema basado en bloqueos las transacciones OLTP y DSS tienen una difícil coexistencia. En la práctica, un sistema de este tipo debe “desconectar” la base de datos para poder efectuar la copia de seguridad (“Su operación no puede efectuarse en estos momentos”, me dice la verde pantalla de mi cajero). De hecho, uno de los objetivos de técnicas como la replicación es el poder aislar físicamente a las aplicaciones de estos dos tipos entre sí. Sin embargo, InterBase no tiene ninguna dificultad para permitir el uso consistente y simultáneo de ambos tipos de transacciones. Esta clase de consideraciones condiciona muchas veces el rendimiento de un sistema de bases de datos.

En contraste con los sistemas basados en bloqueos, que necesitan un *log file* para el control de atomicidad, en una arquitectura como la de InterBase, las propias versiones de las filas modificadas son la única estructura necesaria para garantizar la atomicidad. En realidad, hasta que una transacción finaliza, las versiones modificadas representan datos “tentativos”, no incorporados a la estructura principal de la base de datos. Si la base de datos falla durante una transacción, basta con reiniciar el sistema para tener la base de datos en un estado estable, el estado del que nunca salió. Las versiones tentativas se convierten en “basura”, que se puede recoger y reciclar.



Y este es el inconveniente, un inconveniente menor en mi opinión, de la arquitectura multigeneracional: la necesidad de efectuar de vez en cuando una operación de recogida de basura (*garbage collection*). Esta operación se activa periódicamente en InterBase cada cierto número elevado de transacciones, y puede coexistir con otras transacciones que estén ejecutándose simultáneamente. También se puede programar la operación para efectuarla en momentos de poco tráfico en el sistema; la recogida de basura consume, naturalmente, tiempo de procesamiento en el servidor.

Niveles de aislamiento y transacciones implícitas

Antes en este capítulo, hemos mencionado la existencia de transacciones implícitas para los sistemas de bases de datos SQL, que tienen lugar al no iniciarse transacciones explícitas. Es necesario tomar conciencia de este hecho por una pregunta para la que ya podemos dar respuesta: ¿cuál es el nivel de aislamiento de estas transacciones implícitas?

Resulta que el BDE utiliza por omisión el nivel *tiReadCommitted*, que no nos protege de la aparición de lecturas no repetibles. La elección es sensata, pues muchos sistemas no permiten transacciones con lecturas repetibles en las que simultáneamente se pueda escribir en la base de datos (Oracle, por ejemplo).

Pero existe una razón de más peso, aunque menos evidente, para dejar las cosas como están. Como cualquier desarrollador sabe, la mayor parte de una aplicación de bases de datos típica se ocupa de mantenimientos de tablas simples: altas, bajas y modificaciones de registros individuales. Para este modo de trabajo es preferible que aparezcan lecturas no repetibles. Si nos aisláramos en el nivel *tiRepeatableRead* y alguien modificase un registro desde otro puesto, tendríamos que entrar y salir del programa (o iniciar una transacción explícitamente) para poder ver ese cambio.

De hecho, el controlador de InterBase del BDE puede configurarse para que las transacciones implícitas garanticen lecturas repetibles; basta con sumar 512 al valor

del parámetro *DRIVER_FLAGS*. Sin embargo, por las razones que acabo de exponer, esta técnica no es recomendable en situaciones generales.

De modo que si estamos realizando operaciones de mantenimiento sobre una tabla simple, sencillamente utilizamos un nivel de aislamiento intermedio. En definitiva se trata de operaciones que son atómicas de por sí, ¿o no? ¡Cuidado, pues podemos equivocarnos! Pueden existir *triggers* asociados a las instrucciones de actualización sobre la tabla, que modifiquen otros registros paralelamente. En tales casos, hay que analizar detenidamente si el uso de una transacción *read committed* puede ocasionar incoherencias en la base de datos. De ser positiva la respuesta, estaremos obligados a resolver el problema de algún modo: trabajando con transacciones explícitas (Inter-Base) o utilizando mecanismos de bajo nivel en SQL para garantizar la imposición de bloqueos (Oracle, MS SQL Server). Estos mecanismos de bajo nivel serán estudiados en los capítulos que vienen a continuación.

Microsoft SQL Server

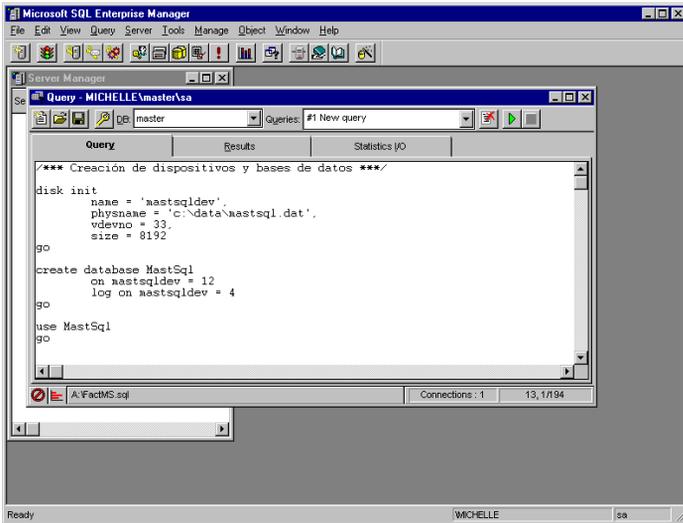
EN ESTE CAPÍTULO ANALIZAREMOS LAS CARACTERÍSTICAS generales de la implementación de SQL por parte de Microsoft SQL Server. Este sistema es uno de los más extendidos en el mundo de las redes basadas en Windows NT. No está, sin embargo, dentro de la lista de mis sistemas favoritos, por dos razones. La primera: una arquitectura bastante pobre, con tamaño fijo de página, bloqueos a nivel de página que disminuyen la capacidad de modificación concurrente, ficheros de datos y de transacciones de tamaño fijo... La segunda razón: MS SQL Server tiene uno de los dialectos de SQL más retorcidos y horribles del mundo de los servidores de datos relacionales, dudoso privilegio que comparte con Sybase.

La primera razón pierde algo de peso con la aparición de la versión 7 de este sistema de bases de datos, que mejora bastante la arquitectura física utilizada. En el presente capítulo describiremos esta versión y la 6.5, debido a su amplia implantación.

Herramientas de desarrollo en el cliente

La herramienta adecuada para diseñar y administrar bases de datos de MS SQL Server se llama *SQL Enterprise Manager*. Normalmente, esta aplicación se instala en el servidor de datos, pero podemos también instalarla en el cliente. Con el Enterprise Manager podemos crear dispositivos (más adelante veremos qué son), bases de datos, crear usuarios y administrar sus contraseñas, e incluso gestionar otros servidores de forma remota.

Si lo que necesitamos es ejecutar consultas individuales o todo un *script* con instrucciones, el arma adecuada es el *SQL Query Tool*, que puede ejecutarse como aplicación independiente, o desde un comando de menú de *SQL Enterprise Manager*, como se muestra en la siguiente imagen. Dentro del menú *File* de esta aplicación encontraremos un comando para que ejecutemos nuestros *scripts* SQL.

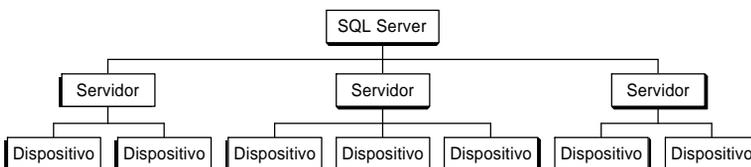


Hay que reconocer que las herramientas de administración de MS SQL Server clasifican entre las amigables con el usuario. Esto se acentúa en la versión 7, en la cual pegas una patada al servidor y saltan cinco o seis asistentes (*wizard*s) para adivinar qué es lo que quieres realmente hacer.

Creación de bases de datos con MS SQL Server

Uno de los cambios que introduce la versión 7 de SQL Server es precisamente en la forma de crear bases de datos. En esta sección, sin embargo, describiré fundamentalmente la técnica empleada hasta la versión 6.5, para todos aquellos que prefieran seguir sufriendo. Además, las bases de datos de Sybase se crean con mecanismos similares.

Para trabajar con SQL Server 6.5 hay que comprender qué es un *dispositivo (device)*. Muy fácil: es un fichero del sistema operativo. ¿Entonces por qué inventar un nombre diferente? La razón está en la prehistoria de este producto, a cargo de Sybase. A diferencia de SQL Server, el servidor de Sybase puede ejecutarse en distintos sistemas operativos, y puede que, en determinado sistema, un fichero físico no sea el soporte más adecuado para el almacenamiento de una base de datos.



Los dispositivos pertenecen al servidor en que se crean. Para crear un dispositivo hay que indicar el nombre del fichero, el tamaño inicial, un nombre lógico y un número entero único, entre 0 y 255, que servirá internamente como identificación. Pueden crearse dispositivos en cualquiera de los discos locales del servidor. La sintaxis completa de la instrucción de creación de dispositivos en SQL Server es la siguiente:

```
disk init
  name='nombre_lógico',
  physname='nombre_de_fichero',
  vdevno=número_de_dispositivo_virtual,
  size=número_de_bloques
  [, vstart=dirección_virtual]
```

Por ejemplo, la siguiente instrucción crea un dispositivo basado en el fichero *mi-disp.dat*, con 20 Mb de espacio:

```
disk init name='MiDisp',
  physname='c:\datos\midisp.dat',
  vdevno=33,
  size=10000
```

Hay que tener en cuenta que el tamaño de bloque es siempre el mismo: 2048 bytes. En SQL Server 7 este tamaño aumenta a 8192 bytes, pero siempre es constante. Otro problema relacionado con los dispositivos consiste en que no aumentan su tamaño por demanda. Si una base de datos situada en uno de estos dispositivos necesita más memoria, hay que ampliar la capacidad del dispositivo de forma manual. Por supuesto, siempre podemos contratar a un “administrador de bases de datos” que se ocupe del asunto, y de esa forma ayudamos a reducir las cifras del paro.

Ahora podemos crear las bases de datos, mediante la siguiente instrucción:

```
create database nombre_base_de_datos
  [on {default|dispositivo} [=tamaño]
  [, dispositivo [=tamaño]]...]
  [log on dispositivo [=tamaño]
  [, dispositivo [=tamaño]]...]
  [for load]
```

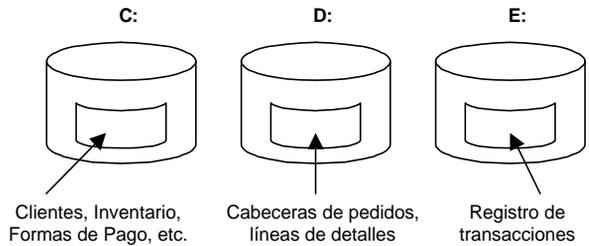
En este caso, el tamaño siempre se especifica en megabytes, como en la siguiente instrucción:

```
create database Facturas on MiDisp=20, log on MiLog=5
```

La base de datos *Facturas* tendrá sus datos en el dispositivo *MiDisp*, mientras que la información sobre transacciones pendientes se almacenará en el segmento *MiLog*. ¿Qué información es ésta? La mayoría de los sistemas de gestión de bases de datos, con la excepción de InterBase, implementan la atomicidad de las transacciones llevando cuenta de los cambios efectuados durante una transacción en un *log file*, o *re-*

gistro de transacciones. Si cancelamos una transacción, la información almacenada en este fichero se utiliza para anular las inserciones, borrados y modificaciones realizadas desde el inicio de la transacción. Es conveniente que el registro de transacciones y los datos residan en dispositivos diferentes, y de ser posible, que estén situados en discos diferentes. De este modo, se puede aprovechar la concurrencia ofrecida por el uso de distintos controladores físicos de discos, a nivel de hardware.

SQL Server 6.5 permite incluso definir varios dispositivos para almacenar datos y para el registro de transacciones. Utilizando instrucciones como *sp_addsegment* y *sp_placeobject*, podemos almacenar distintas tablas en diferentes discos, como se muestra en el siguiente esquema:



Recuerde que el objetivo principal de la segmentación es el aumentar la eficiencia de la base de datos.

Bases de datos en la versión 7

En SQL Server 7 desaparece el concepto de dispositivo, y las bases de datos se deben definir directamente sobre ficheros físicos. El fichero de datos principal tiene la extensión *mdf*, el fichero *log* tiene la extensión *ldf*, y los ficheros de datos secundarios, *ndf*. Los nuevos ficheros pueden crecer por demanda, como sucede en los sistemas de bases de datos "serios". Ya he mencionado, además, que el tamaño de página se aumenta a 8192, aunque este valor no puede modificarse.

He aquí un ejemplo de creación de bases de datos con la nueva versión:

```
create database Ventas on primary
    (name = Datos1, filename = 'c:\ventas\datos1.mdf',
     size = 100MB, maxsize = 200, filegrowth = 20),
    filegroup Secundario
    (name = Datos2, filename = 'd:\ventas\datos2.ndf',
     size = 100MB, maxsize = 200, filegrowth = 20),
    log on
    (name = Log1, filename = 'e:\ventas\log1.ldf',
     size = 100MB, maxsize = 200, filegrowth = 20)
```

El creador de la base de datos anterior tenía a su disposición tres discos duros diferentes (no tienen sentido aquí las particiones lógicas). Su fichero de datos por omisión, indicado mediante la palabra reservada **primary**, se sitúa en el disco C, tiene 100MB inicialmente, puede crecer hasta 200MB, y cada vez que crece añade 20MB al tamaño del fichero.

En el segundo disco crea otro fichero de datos, pero esta vez utiliza la cláusula **filegroup** para darle un nombre lógico. Realmente, *Datos2* es un nombre lógico que se refiere al fichero, pero los grupos de ficheros permiten agrupar lógicamente varios ficheros físicos. ¿Con qué objetivo? Normalmente, si se utilizan varios ficheros para una base de datos, el segundo fichero no se utiliza hasta que se acabe la memoria del primer fichero. Sin embargo, si agrupamos los ficheros mediante **filegroup** el crecimiento ocurre de forma proporcional al tamaño relativo de los ficheros. Además, el nombre de los grupos de ficheros se utiliza en las instrucciones de creación de índices y tablas para indicar dónde se ubicarán estos objetos. Ya no se pueden utilizar los antiguos procedimientos *sp_addsegment* y *sp_placeobject*.

Por último, el fichero *log* se ha creado en el tercer disco disponible.

Tipos de datos predefinidos

Los tipos de datos soportados por SQL Server son los siguientes:

Tipo de datos	Implementación de SQL Server
Binario	<i>binary</i> [(n)], <i>varbinary</i> [(n)]
Caracteres	<i>char</i> [(n)], <i>varchar</i> [(n)]
Fecha y hora	<i>datetime</i> , <i>smalldatetime</i>
Numérico exacto	<i>decimal</i> [(p[, s])], <i>numeric</i> [(p[, s])]
Numérico aproximado	<i>float</i> [(n)], <i>real</i>
Enteros	<i>int</i> , <i>smallint</i> , <i>tinyint</i>
Moneda	<i>money</i> , <i>smallmoney</i>
Especiales	<i>bit</i> , <i>timestamp</i>
Texto e imágenes	<i>text</i> , <i>image</i>

Nos encontramos aquí con el mismo problema que en InterBase: el tipo *datetime* representa simultáneamente fechas y horas. Curiosamente, el tipo *timestamp* no se refiere a fechas, sino que es un entero de 8 bytes que se actualiza automáticamente cuando se crea o se modifica una fila. El tipo *smalldatetime* tiene menos precisión que *datetime*. Almacena la fecha como el número de días a partir del 1 de enero del 1900, utilizando dos bytes, y la hora como el número de minutos a partir de media noche. Por lo tanto, solamente le será útil si su programa debe quedar fuera de circulación antes del 6 de junio del 2079. Si es usted uno de los Inmortales, o le preocupa el futuro del Planeta, no lo utilice. Del mismo modo, *smallmoney* tiene un rango que va

desde -214.748,3648 hasta aproximadamente el mismo número positivo. Ya sean euros o dólares, no alcanza para representar la facturación mensual de The Coca-Cola Company.

Aunque el tipo *nvarchar* de SQL Server permite almacenar cadenas de caracteres de longitud variable, los registros que contienen estos campos siguen ocupando un tamaño fijo, al menos hasta la versión 6.5. Al parecer, la versión 7 corrige este derrochador comportamiento.

SQL Server 7 añade varios tipos a la colección presentada. El más interesante es el tipo **uniqueidentifier**, que contiene valores enteros de 128 bits generados por el sistema. A este tipo de números se les denomina *GUID*, por *Global Unique Identifier*, y nos tropezaremos con ellos al estudiar el modelo COM. Lamentablemente, la interfaz *DBLibrary*, que es la que utiliza el BDE para acceder a SQL Server, no ofrece soporte para los nuevos tipos de datos, transmitiendo este problema a C++ Builder. Habrá que esperar a que Borland desarrolle un acceso a SQL Server mediante OLE DB.

Tipos de datos definidos por el programador

Para no variar, MS SQL Server complica y distorsiona la creación de tipos de datos por el programador. En vez de utilizar dominios, o algún mecanismo elegante de definición, se utiliza un procedimiento almacenado, *sp_addtype*, para crear nuevos tipos de datos:

```
sp_addtype telefono, 'char(9)', null
```

Estas son todas las posibilidades de *sp_addtype*: especificar el nombre del nuevo tipo, indicar a qué tipo predefinido es equivalente, y decir si el tipo admite valores nulos o no. ¿Existe alguna forma de asociar una restricción a estos tipos de datos? Sí, pero la técnica empleada es digna de Forrest Gump. Primero hay que crear una *regla*:

```
create rule solo_numeros as
    @valor not like '%[^0-9]%'
```

Luego, hay que *asociar* la regla al tipo de datos:

```
sp_bindrule solo_numeros, telefono
```

Existen también instrucciones **create default** y *sp_binddefault* para asociar valores por omisión a tipos de datos creados por el programador.

Al igual que sucede con los dominios de InterBase, el Diccionario de Datos del BDE utiliza los tipos de datos definidos por el usuario de SQL Server 7 para sacar factor común en la creación de conjuntos de atributos. El Diccionario de Datos se estudiará en el capítulo 19.

Creación de tablas y atributos de columnas

La sintaxis para la creación de tablas, a grandes rasgos, es similar a la de InterBase, con la lógica adición de una cláusula opcional para indicar un segmento donde colocar la tabla:

```
create table Clientes (
    Codigo          int not null primary key,
    Nombre          varchar(30) not null unique,
    Direccion       varchar(35) null,
)
on SegmentoMaestro
```

En MS SQL Server 7, la cláusula **on** se refiere a un grupo de ficheros definido durante la creación de la base de datos o posteriormente. Recuerde que en la nueva versión ya no existen los segmentos.

Sin embargo, hay diferencias notables en los atributos que se especifican en la definición de columnas. Por ejemplo, en la instrucción anterior he indicado explícitamente que la columna *Direccion* admite valores nulos. ¿Por qué? ¿Acaso no es éste el comportamiento asumido por omisión en SQL estándar? Sí, pero Microsoft no está de acuerdo con esta filosofía, y asume por omisión que una columna no puede recibir nulos. Peor aún: la opción '*ANSI null default*', del procedimiento predefinido de configuración *sp_dboption*, puede influir en qué tipo de comportamiento se asume. ¿Le cuesta tanto a esta compañía respetar un estándar?

Uno de los recursos específicos de MS SQL Server es la definición de columnas con el atributo **identity**. Por ejemplo:

```
create table Colores (
    Codigo          integer identity(0,1) primary key,
    Descripcion     varchar(30) not null unique
)
```

En la tabla anterior, el valor del código del color es generado por el sistema, partiendo de cero, y con incrementos de uno en uno. Sin embargo, no es una opción recomendable, a mi entender, si realizamos inserciones desde C++ Builder en esa tabla y estamos mostrando sus datos en pantalla. El BDE tiene problemas para releer registros cuando su clave primaria se asigna o se modifica en el servidor, como en

este caso. Sucede lo mismo que con los generadores de InterBase y las secuencias de Oracle.

Las cláusulas **check** de SQL Server permiten solamente expresiones que involucran a las columnas de la fila actual de la tabla, que es lo que manda el estándar (¡por una vez!). Esta vez podemos decir algo a favor: las expresiones escalares de SQL Server son más potentes que las de InterBase, y permiten diseñar validaciones consecuentemente más complejas:

```
create table Clientes (
    /* ... */
    Telefono      char(11),
    check (Telefono like
          '[0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9][0-9][0-9]' or
          Telefono like
          '[0-9][0-9]-[0-9][0-9]-[0-9][0-9][0-9][0-9]')
```

Una opción interesante de SQL Server es la posibilidad de crear tablas temporales. Para esto, basta con añadir uno de los prefijos # ó ## al nombre de la tabla. En el primer caso, la tabla temporal solamente será visible para el usuario que la crea, y será eliminada en cuanto el usuario cierre su conexión activa. En el segundo caso, se trata de una tabla temporal global, que desaparecerá en cuanto el último usuario que la esté usando se desconecte.

Integridad referencial

Las versiones de este producto anteriores a la 7 realizan una pobre implementación de las restricciones de integridad referencial. En contraste, la versión 7 realiza una pobre implementación de las restricciones de integridad referencial. Es decir, que al parecer todo cambia para seguir igual.

¿De qué me estoy quejando? Principalmente, de que SQL Server no define ningún tipo de acción referencial, aparte de la prohibición de borrados y de actualizaciones en la clave de la fila maestra. Y, para agravar el problema, tenemos la complicación adicional del tipo único de *triggers* permitido por el sistema, que se dispara *después* de la operación. De este modo, la simple adición de una propagación de borrados en cascada requiere que la restricción de integridad referencial se anule y se implemente totalmente “a mano”. Más adelante veremos cómo.

MS SQL Server no define automáticamente los índices secundarios necesarios en la tabla dependiente. El programador debe crearlos explícitamente. Además, la única acción referencial permitida es la prohibición de borrados y actualizaciones.

Indices

La sintaxis para la creación de índices en MS SQL Server es básicamente la siguiente:

```
create [unique] [clustered | nonclustered] index Índice
on Tabla (Columna [, Columna ...])
[on GrupoFicheros]
```

Tenemos la posibilidad de definir *índices agrupados* (*clustered indexes*). Cuando una tabla tiene un índice agrupado, sus registros se almacenan ordenados físicamente en las páginas de datos de la base de datos. Esto implica que solamente puede haber un índice agrupado por tabla. Es muy eficiente recorrer una tabla utilizando uno de estos índices. En cambio, puede ser relativamente costoso mantener el índice agrupado cuando se producen actualizaciones.

Como sabemos, la definición de una clave primaria o única genera automáticamente un índice. Si queremos que éste sea el índice agrupado de la tabla, podemos indicarlo de la siguiente manera:

```
create table Clientes (
    Codigo          int not null,
    Nombre          varchar(30) not null,
    /* ... */
    primary key (Codigo),
    unique clustered (Nombre)
)
```

Los índices de MS SQL Server, a diferencia de los de InterBase y Oracle, pueden ignorar la distinción entre mayúsculas y minúsculas. Sin embargo, este comportamiento debe establecerse durante la instalación del servidor, y no puede modificarse más adelante.

Seguridad en MS SQL Server

SQL Server implementa un sistema de seguridad en dos niveles bastante complicado. En el primer nivel, se definen los *logins*, que se utilizan para conectar con determinado servidor. Hasta cierto punto, estos *logins* son similares a los “usuarios” de InterBase, pues se asocian con el servidor, no con la base de datos. La validación de las contraseñas puede estar a cargo del propio SQL Server, o de Windows NT, y en este último caso se habla de *seguridad integrada*. Con la seguridad integrada, los *logins* corresponden a usuarios del sistema operativo. Inicialmente, una instalación de SQL Server define el *login* del administrador como *sa*, sin contraseña.

En una segunda fase, al nivel ya de las bases de datos, se definen usuarios, hablando con propiedad. Los *logins* definidos en la fase anterior se asocian entonces con usua-

rios de una base de datos, y es a estos usuarios a los que se otorgan privilegios, mediante nuestras viejas y conocidas instrucciones **grant** y **revoke**. Al crearse una base de datos se define automáticamente un usuario especial como propietario de la misma, que se identifica como *dbo*: (*database owner*), y que tendrá acceso total a todos los objetos creados dentro de esa base de datos, ya sea por él mismo o por otro usuario. Por supuesto, el administrador del sistema también tiene acceso sin restricciones a esos objetos.

Al desplegar la lista de posibles valores de la propiedad *TableName* de un componente de tablas en C++ Builder, las tablas existentes aparecen cualificadas con el nombre del usuario que las ha creado, que en MS SQL Server es generalmente el *dbo*. Lo mismo sucede en Oracle, aunque no en InterBase. Este comportamiento compromete la portabilidad de la aplicación entre distintos servidores, pero no es aconsejable eliminar el prefijo de usuario del nombre de la tabla. En tal caso, al BDE le resultará difícil consultar el catálogo de la base de datos para encontrar información sobre índices. Los errores que se producen como consecuencia son bastante perversos e insidiosos: cierta aplicación escrita en Delphi 3, que tuve que corregir, funcionaba perfectamente, hasta que alguien cambiaba dinámicamente el índice activo. Al concluir la ejecución, se producía un misterioso fallo de protección general. Por supuesto, todo se arregló al introducir los prefijos de usuarios en los nombres de tablas.

Procedimientos almacenados

He aquí la sintaxis de la creación de procedimientos almacenados en Transact-SQL:

```
create procedure Nombre[;Numero] [Parámetros]
    [for replication|with recompile [with encryption]]
as Instrucciones
```

Por ejemplo:

```
create procedure ProximoCodigo @cod int output as
begin
    select @cod = ProxCod
    from Numeros holdlock
    update Numeros
    set ProxCod = ProxCod + 1
end
```

ProximoCodigo es el típico procedimiento almacenado que extrae un valor numérico de una tabla de contadores y lo devuelve al cliente que lo ha solicitado. En primer lugar, vemos que los parámetros y variables de Transact-SQL deben ir precedidos obligatoriamente por un signo @. Esto ya es una molestia, porque el convenio de nombres de parámetros es diferente al de Oracle e InterBase. Si desarrollamos una

aplicación que deba trabajar indistintamente con cualquiera de estos servidores, habrá que considerar el caso especial en que el servidor sea MS SQL, pues los nombres de parámetros se almacenan estáticamente en el fichero *d/m*.

Además de ciertas curiosidades sintácticas, como que las instrucciones no necesitan ir separadas por puntos y comas, lo que más llama la atención en el procedimiento anterior es el uso de la opción **holdlock** en la cláusula **from** de la primera instrucción. Esta opción fuerza a SQL Server a mantener un bloqueo de lectura sobre la fila seleccionada, al menos hasta que finalice la transacción actual, y permite evitar tener que configurar las transacciones para el nivel de lecturas repetibles.

Vemos también que Transact-SQL no utiliza la cláusula **into** de InterBase y Oracle para asignar los resultados de un **select** a variables o parámetros, sino que incorpora asignaciones en la propia cláusula **select**. De hecho, el que no exista un separador de instrucciones nos obliga a anteponer la palabra reservada **select** delante de una simple asignación de variables:

```
declare @i integer           /* Declaramos una variable local */
select @i = @@rowcount      /* Le asignamos una global */
if (@i > 255)               /* Preguntamos por su valor */
    /* ... */
```

Es característico de MS SQL Server y Sybase la posibilidad de programar procedimientos que devuelvan un conjunto de datos. En tal caso, el cuerpo del procedimiento debe consistir en una sentencia **select**, que puede contener parámetros. Sin embargo, la misma funcionalidad se logra desde C++ Builder con consultas paramétricas, que no comprometen además la portabilidad de la aplicación.

Cursores

MS SQL Server no implementa la sentencia **for...do** de InterBase. En realidad, esa instrucción es única para InterBase, y casi todos los demás servidores ofrecen *cursores* como mecanismo de recorrido sobre tablas. Un cursor se define asociando un nombre a una instrucción SQL, como en este ejemplo:

```
declare QueHaHechoEsteTio cursor for
select Fecha, Total from Pedidos
where RefEmpleado = (select Codigo from Empleados
                    where Nombre = @Nombre)
```

Observe que estamos utilizando una variable, *@Nombre*, en la definición del cursor. Se supone que esa variable está disponible en el lugar donde se declara el cursor. Cuando se realiza la declaración no suenan trompetas en el cielo ni tiembla el disco

duro; es solamente eso, una declaración. Cuando sí ocurre algo es al ejecutarse la siguiente instrucción:

```
open QueHaHechoEsteTio
```

Ahora se abre el cursor y queda preparado para su recorrido, que se realiza de acuerdo al siguiente esquema:

```
declare @Fecha datetime, @Total integer
fetch from QueHaHechoEsteTio into @Fecha, @Total
while (@@fetch_status = 0)
begin
    /* Hacer algo con las variables recuperadas */
    fetch from QueHaHechoEsteTio into @Fecha, @Total
end
```

La variable global `@@fetch_status` es de vital importancia para el algoritmo, pues deja de valer cero en cuanto el cursor llega a su fin. Tome nota también de que hay que ejecutar un **fetch** también antes de entrar en el bucle **while**.

Una vez que se ha terminado el trabajo con el cursor, es necesario cerrarlo por medio de la instrucción **close** y, en ocasiones, liberar las estructuras asociadas, mediante la instrucción **deallocate**:

```
close QueHaHechoEsteTio
deallocate QueHaHechoEsteTio
```

La diferencia entre **close** y **deallocate** es que después de ejecutar la primera, aún podemos reabrir el cursor. En cambio, después de ejecutar la segunda, tendríamos que volver a declarar el cursor con **declare**, antes de poder utilizarlo.

El siguiente ejemplo, un poco más complicado, cumple la misma función que un procedimiento almacenado de mismo nombre que hemos desarrollado en el capítulo sobre InterBase, y que estaba basado en la instrucción **for...do**. Su objetivo es recorrer ordenadamente todas las líneas de detalles de un pedido, y actualizar consecuentemente las existencias en el inventario:

```
create procedure ActualizarInventario @Pedido integer as
begin
    declare dets cursor for
        select RefArticulo, Cantidad
        from Detalles
        where RefPedido = @Pedido
        order by RefArticulo
    declare @CodArt integer, @Cant integer

    open dets
    fetch next from dets into @CodArt, @Cant
    while (@@fetch_status = 0)
    begin
```

```

update Articulos
set Pedidos = Pedidos + @Cant
where Codigo = @CodArt
fetch next from dets into @CodArt, @Cant
end
close dets
deallocate dets
end

```

Microsoft ofrece cursores bidireccionales en el servidor, y están muy orgullosos de ellos. Vale, los cursores bidireccionales están muy bien. Felicidades. Lástima que el BDE no los pueda aprovechar (es culpa del BDE). Y que C++ Builder sea tan bueno que no merezca la pena cambiar de herramienta de desarrollo.

Triggers en Transact-SQL

Los *triggers* que implementa Transact-SQL, el lenguaje de programación de MS SQL Server, son muy diferentes a los de InterBase y a los de la mayoría de los servidores existentes. Esta es la sintaxis general de la operación de creación de *triggers*:

```

create trigger NombreTrigger on Tabla
[with encryption]
for {insert,update,delete}
[with append] [not for replication]
as InstruccionSQL

```

En primer lugar, cada tabla solamente admite hasta tres *triggers*: uno para cada una de las operaciones **insert**, **update** y **delete**. Sin embargo, un mismo *trigger* puede dispararse para dos operaciones diferentes sobre la misma tabla. Esto es útil, por ejemplo, en *triggers* que validan datos en inserciones y modificaciones.

SQL Server 7 ha corregido esta situación, y permite definir más de un *trigger* por evento. Incluso si hemos activado la compatibilidad con la versión 6.5 por medio del procedimiento *sp_dbcmplevel*, la cláusula **with append** indica que el nuevo *trigger* se debe añadir a la lista de *triggers* existentes para el evento, en vez de sustituir a uno ya creado. De no estar activa la compatibilidad, dicha cláusula no tiene efecto alguno.

Pero la principal diferencia consiste en el momento en que se disparan. Un *trigger* decente debe dispararse antes o después de una operación sobre *cada fila*. Los de Transact-SQL, en contraste, se disparan solamente *después* de una instrucción, que puede afectar a *una* o *más* filas. Por ejemplo, si ejecutamos la siguiente instrucción, el posible *trigger* asociado se disparará únicamente cuando se hayan borrado todos los registros correspondientes:

```
delete from Clientes
where Planeta <> "Tierra"
```

El siguiente ejemplo muestra como mover los registros borrados a una tabla de copia de respaldo:

```
create trigger GuardarBorrados
on Clientes
for delete as
insert into CopiaRespaldo select * from deleted
```

Como se puede ver, para este tipo de *trigger* no valen las variables *old* y *new*. Se utilizan en cambio las tablas *inserted* y *deleted*:

	insert	delete	update
<i>inserted</i>	Sí	No	Sí
<i>deleted</i>	No	Sí	Sí

Estas tablas se almacenan en la memoria del servidor. Si durante el procesamiento del *trigger* se realizan modificaciones secundarias en la tabla base, no vuelve a activarse el *trigger*, por razones lógicas.

Como es fácil de comprender, es más difícil trabajar con *inserted* y *deleted* que con las variables *new* y *old*. El siguiente *trigger* modifica las existencias de una tabla de inventarios cada vez que se crea una línea de pedido:

```
create trigger NuevoDetalle on Detalles for insert as
begin
    if @@RowCount = 1
        update Articulos
        set Pedidos = Pedidos + Cantidad
        from Inserted
        where Articulos.Codigo = Inserted.RefArticulo
    else
        update Articulos
        set Pedidos = Pedidos +
            (select sum(Cantidad)
             from Inserted
             where Inserted.RefArticulo=Articulos.Codigo)
        where Codigo in
            (select RefArticulo
             from Inserted)
end
```

La variable global predefinida *@@RowCount* indica cuántas filas han sido afectadas por la última operación. Al preguntar por el valor de la misma en la primera instrucción del *trigger* estamos asegurándonos de que el valor obtenido corresponde a la instrucción que desencadenó su ejecución. Observe también la sintaxis peculiar de la primera de las instrucciones **update**. La instrucción en cuestión es equivalente a la siguiente:

```

update Articulos
set Pedidos = Pedidos + Cantidad
from Inserted
where Articulos.Codigo =
    (select RefArticulo
     from Inserted) /* Singleton select! */

```

¿Hasta qué punto nos afecta la mala conducta de los *triggers* de Transact-SQL? La verdad es que muy poco, si utilizamos principalmente los métodos de tablas del BDE para actualizar datos. El hecho es que los métodos de actualización del BDE siempre modifican una sola fila por instrucción, por lo que *@@RowCount* siempre será uno para estas operaciones. En este *trigger*, un poco más complejo, se asume implícitamente que las inserciones de pedidos tienen lugar de una en una:

```

create trigger NuevoPedido on Pedidos for insert as
begin
    declare @UltimaFecha datetime, @FechaVenta datetime,
            @Num int, @CodPed int, @CodCli int
    select @CodPed = Codigo, @CodCli = RefCliente,
            @FechaVenta = FechaVenta
    from inserted
    select @Num = ProximoNumero
    from Numeros holdlock
    update Numeros
    set ProximoNumero = ProximoNumero + 1
    update Pedidos
    set Numero = @Num
    where Codigo = @CodPed
    select @UltimaFecha = UltimoPedido
    from Clientes
    where Codigo = @CodCli
    if (@UltimaFecha < @FechaVenta)
        update Clientes
        set UltimoPedido = @FechaVenta
        where Codigo = @CodCli
end

```

Observe cómo hemos vuelto a utilizar **holdlock** para garantizar que no hayan huecos en la secuencia de valores asignados al número de pedido.

Integridad referencial mediante *triggers*

Es bastante complicado intentar añadir borrados o actualizaciones en cascada a las restricciones de integridad referencial de MS SQL Server. Como los *triggers* se ejecutan al final de la operación, antes de su ejecución se verifican las restricciones de integridad en general. Por lo tanto, si queremos implementar un borrado en cascada tenemos que eliminar la restricción que hemos puesto antes, y asumir también la verificación de la misma.

Partamos de la relación existente entre cabeceras de pedidos y líneas de detalles, y supongamos que no hemos declarado la cláusula **foreign key** en la declaración de esta última tabla. El siguiente *trigger* se encargaría de comprobar que no se inserte un detalle que no corresponda a un pedido existente, y que tampoco se pueda modificar posteriormente esta referencia a un valor incorrecto:

```
create trigger VerificarPedido on Detalles for update, insert as
  if exists(select * from Inserted
            where Inserted.RefPedido not in
                  (selectCodigo from Pedidos))
begin
  raiserror('Código de pedido incorrecto', 16, 1)
  rollback tran
end
```

Aquí estamos introduciendo el procedimiento *raiserror* (sí, con una sola 'e'), que sustituye a las excepciones de InterBase. El primer argumento es el mensaje de error. El segundo es la severidad; si es 10 o menor, no se produce realmente un error. En cuanto al tercero (un código de estado), no tiene importancia para SQL Server en estos momentos. A continuación de la llamada a esta instrucción, se deshacen los cambios efectuados hasta el momento y se interrumpe el flujo de ejecución. Recuerde que esto en InterBase ocurría automáticamente.

La documentación de SQL Server recomienda como posible alternativa que el *trigger* solamente deshaga los cambios incorrectos, en vez de anular todas las modificaciones. Pero, en mi humilde opinión, esta técnica puede ser peligrosa. Prefiero considerar atómicas a todas las operaciones lanzadas desde el cliente: que se ejecute todo o nada.

El *trigger* que propaga los borrados es sencillo:

```
create trigger BorrarPedido on Pedidos for delete as
  delete from Detalles
  where RefPedido in (selectCodigo from Deleted)
```

Sin embargo, el que detecta la modificación de la clave primaria de los pedidos es sumamente complicado. Cuando se produce esta modificación, nos encontramos de repente con dos tablas, *inserted* y *deleted*. Si solamente se ha modificado un registro, podemos establecer fácilmente la conexión entre las filas de ambas tablas, para saber qué nuevo valor corresponde a qué viejo valor. Pero si se han modificado varias filas, esto es imposible en general. Así que vamos a prohibir las modificaciones en la tabla maestra:

```
create trigger ModificarPedido on Pedidos for update as
  if update(Codigo)
begin
  raiserror('No se puede modificar la clave primaria',
           16, 1)
```

```

        rollback tran
    end

```

Le dejo al lector que implemente la propagación de la modificación en el caso especial en que ésta afecta solamente a una fila de la tabla maestra.

Triggers anidados y triggers recursivos

¿Qué sucede si durante la ejecución de un *trigger* se modifica alguna otra tabla, y esa otra tabla tiene también un *trigger* asociado? Todo depende de cómo esté configurado el servidor. Ejecute el siguiente comando en el programa *Query Analyzer*:

```

sp_configure 'nested triggers'
go

```

El procedimiento devolverá el valor actual de dicha opción, que puede ser *0* ó *1*. Si está activa, el *trigger* de la tabla afectada secundariamente también se dispara. El número de niveles de anidamiento puede llegar a un máximo de 16. Para cambiar el valor de la opción, teclee lo siguiente:

```

-- Activar los triggers anidados
sp_dbconfigure 'nested triggers', '1'
go

```

Recuerde que la opción *nested triggers* afecta a todas las bases de datos de un mismo servidor, así que tenga mucho cuidado con lo que hace.

Sin embargo, es muy diferente lo que sucede cuando la tabla modificada por el *trigger* es la propia tabla para la cual se define éste. Una de las consecuencias de que los *triggers* de SQL Server se disparen después de terminada la operación, es que si queremos modificar automáticamente el valor de alguna columna estamos obligados a ejecutar una instrucción adicional sobre la tabla: ya no tenemos una conveniente variable de correlación *new*, que nos permite realizar este cambio antes de la operación:

```

create trigger MantenerVersionMayusculas on Clientes
    for insert, update as
if update(Nombre)
    update Clientes
    set     NombreMay = upper(Nombre)
    where  Codigo in (select Codigo from inserted)

```

Si modificamos el nombre de uno o más clientes, la versión en mayúsculas del nombre de cliente debe actualizarse mediante una segunda instrucción **update**. ¿Y ahora qué, se vuelve a disparar el *trigger*? No si la versión de SQL Server es la 6.5. Pero si

estamos trabajando con la 7, volvemos a depender del estado de una opción de la base de datos que, esta vez, se modifica mediante *sp_dboption*:

```
-- Activar los triggers recursivos  
sp_dboption 'facturacion', 'recursive triggers', 'true'
```

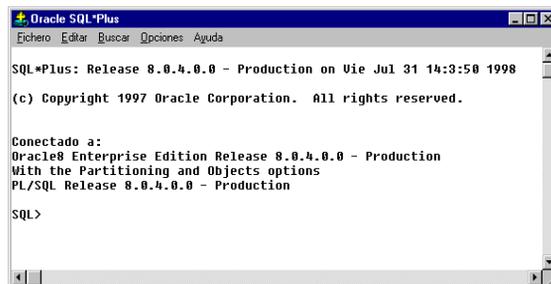
En cualquier caso, si el *trigger* vuelve a ejecutarse de forma recursiva no pasa nada malo en el ejemplo anterior, pues en la segunda ejecución la columna modificada es *NombreMay*, en vez de *Nombre*. Pero hay que tener mucho cuidado con otros casos, en que puede producirse una recursión potencialmente infinita. SQL Server 7 también limita a 16 los niveles de anidamiento de un *trigger* recursivo.

LAS REGLAS DEL JUEGO ESTÁN CAMBIANDO poco a poco, mientras el mundo gira (y mi guitarra solloza). Oracle ha sido el primero de los grandes sistemas relacionales en incluir extensiones orientadas a objetos significativas. Realmente, Oracle siempre ha destacado por su labor innovadora en relación con su modelo de datos y el lenguaje de programación en el servidor. De hecho, PL/SQL puede tomarse perfectamente como referencia para el estudio de *triggers*, procedimientos almacenados, tipos de datos, etc.

Por supuesto, no puedo cubrir todo Oracle en un solo capítulo. Por ejemplo, evitaré en lo posible los temas de configuración y administración. Tampoco entraremos en la programación de *packages* y otras técnicas particulares de este sistema, por entender que hacen difícil la posterior migración a otras bases de datos. Sin embargo, sí veremos algo acerca de las extensiones de objetos, debido al soporte que C++ Builder 4 ofrece para las mismas.

Sobreviviendo a SQL*Plus

Oracle ofrece varias utilidades mediante las cuales pueden crearse tablas, procedimientos y tipos en sus bases de datos. La más utilizada se llama *SQL*Plus*, y permite la ejecución de comandos aislados y de *scripts*. La siguiente imagen muestra a SQL*Plus en funcionamiento:



```
Oracle SQL*Plus
Archivo  Editar  Buscar  Opciones  Ayuda

SQL*Plus: Release 8.0.4.0.0 - Production on Vie Jul 31 14:3:50 1998

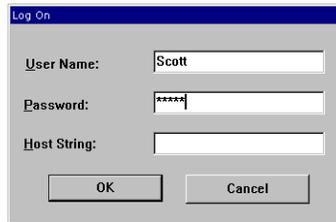
(c) Copyright 1997 Oracle Corporation. All rights reserved.

Conectado a:
Oracle8 Enterprise Edition Release 8.0.4.0.0 - Production
With the Partitioning and Objects options
PL/SQL Release 8.0.4.0.0 - Production

SQL>
```

Como podemos apreciar, es una utilidad basada en el modo texto, como en los viejos tiempos de UNIX, MSDOS y similares. En lógica consecuencia, SQL*Plus tiene fama de incómoda e insoportable, por lo que tenemos que aprovechar al máximo las pocas facilidades que nos ofrece.

Para comenzar, pongamos por caso que nos hemos conectado a la base de datos de ejemplos que trae Oracle como el usuario *Scott* con su contraseña *tiger*. Esta conexión se realiza mediante el cuadro de diálogo que presenta inicialmente SQL*Plus:



Al tratarse, en mi caso, de Personal Oracle, puedo dejar vacío el campo *Host String*. Si desea conectarse a un servidor remoto, aquí debe indicar un nombre de “servicio” creado con SQL*Net Easy Configuration. Más adelante podemos utilizar estas instrucciones para conectarnos como otro usuario, o para desconectarnos:

```
connect system/manager;  
disconnect;
```

System es el nombre del administrador de bases de datos, y *manager* es su contraseña inicial.

En principio, podemos teclear cualquier sentencia SQL en respuesta a la petición (*prompt*) de SQL*Plus, terminándola con un punto y coma. La instrucción puede ocupar varias líneas; en tal caso, en el editor aparecen los números de líneas en la medida en que vamos creándolas. Al final, SQL*Plus detecta el punto y coma, para ejecutar entonces la sentencia:

```
SQL> create table prueba (  
1   Id      integer,  
2   Nombre  varchar(30)  
3 );
```

¿Y qué pasa si hemos metido garrafalmente la extremidad inferior al teclear? Nada, que siempre hay una segunda oportunidad. Teclee *edit*, y aparecerá el Bloc de Notas con el texto de la última sentencia introducida, para que pueda ser corregida. En realidad, el texto se guarda en un fichero temporal, de nombre *afiedt.buf*, y hay que utilizar el siguiente comando para ejecutar el contenido de este fichero, una vez corregido y salvado:

```
SQL> @afiedt.buf
```

El signo @ sirve para ejecutar *scripts* con sentencias PL/SQL. En el ejemplo anterior, no hemos tenido que indicar el directorio donde se encuentra el fichero, pues éste se ha creado en el directorio raíz de Oracle, *c:\orawin95\bin* en mi instalación.

Un comportamiento curioso de SQL*Plus, y que me ha ocasionado no pocos quebraderos de cabeza, es que no permite líneas en blanco dentro de una instrucción. A mí me gusta, por ejemplo, dejar una línea en blanco dentro de la sentencia **create table** para separar la definición de columnas de la definición de restricciones a nivel de tabla. SQL*Plus me obliga a utilizar al menos un comentario en esa posición:

```
create table prueba (
    Id      integer,
    Nombre  varchar(30),
    --
    primary key (Id),
    unique (Nombre)
);
```

Hablemos acerca de los errores. Cuando cometemos alguno gordo, de los de verdad, SQL*Plus lo indica inmediatamente. Pero a veces se comporta solapadamente, casi siempre cuando creamos *triggers*, procedimientos almacenados y objetos similares. Obtenemos de repente este mensaje:

```
Procedure created with compilation errors.
```

No hay misterio: Oracle ha detectado un error, pero es tan listo que ha podido corregirlo él mismo (o al menos eso pretende). En cualquier caso, teclee el comando *show errors* para averiguar cuáles han sido los errores detectados en la última instrucción.

Cuando estamos creando tipos, procedimientos, triggers y otros objetos complejos en una base de datos, debemos utilizar instrucciones que terminan en punto y coma. En este caso, SQL*Plus requiere que terminemos toda la instrucción con una línea que contenga una barra inclinada. Este carácter actúa entonces de forma similar al carácter de terminación de los *scripts* de InterBase.

Instancias, bases de datos, usuarios

En dependencia del sistema operativo donde se ejecuta un servidor de Oracle, éste puede trabajar simultáneamente con una o más bases de datos. En el caso de Personal Oracle para Windows 95, sólo puede estar activa una base de datos a la vez. Pero las versiones completas del producto sí admiten varias bases de datos activas simultáneamente.

A cada base de datos activa, junto a los procesos que Oracle lanza para su mantenimiento y los datos de las conexiones de usuarios, se le denomina *instancia* de Oracle. Cada instancia se identifica con un nombre; en el caso de Oracle Enterprise Edition, la instancia que se asocia a la base de datos instalada por omisión se llama *ORCL*. Si tenemos que mantener simultáneamente varias bases de datos en un mismo servidor, necesitaremos una instancia debidamente identificada para cada una de ellas.

Los administradores de bases de datos tienen el privilegio de poder crear nuevas bases de datos. La instrucción necesaria está cargada de parámetros, como corresponde a una arquitectura compleja y con una larga historia. Este es un ejemplo sencillo de creación de bases de datos, con la mayoría de los parámetros asumidos por omisión:

```
create database Prueba
  datafile 'p_datos' size 10M
  logfile group 1 ('p_log1a', 'p_log1b') size 500K
  group 2 ('p_log2a', 'p_log2b') size 500K
```

Por supuesto, Oracle ofrece herramientas gráficas para crear y modificar bases de datos. En el ejemplo anterior se ha creado una base de datos con un solo fichero de datos, de 10MB, y con dos grupos de ficheros para el registro de transacciones, cada grupo con 500KB.

Cada base de datos de Oracle tiene una lista independiente de usuarios autorizados a trabajar con ella. La siguiente instrucción sirve para crear un nuevo usuario:

```
create user Nombre
  identified [by Contraseña | externally]
  [OpcionesDeUsuario]
```

Al igual que sucede en InterBase 5, se pueden definir roles, y asignarlos a usuarios existentes para simplificar la posterior administración de privilegios con las dos conocidas instrucciones **grant** y **revoke**.

Tipos de datos

Los tipos de datos básicos de Oracle son los siguientes:

Tipo de dato	Significado
<i>varchar2(n)</i> , <i>nvarchar2(n)</i>	Cadenas de caracteres de longitud variable
<i>char(n)</i> , <i>nchar(n)</i>	Cadenas de caracteres de longitud fija
<i>number(p,s)</i>	Números, con escala y precisión
<i>date</i>	Fecha y hora, simultáneamente
<i>long</i>	Cadenas de caracteres de hasta 2GB
<i>raw(n)</i> , <i>long raw</i>	Datos binarios

Tipo de dato	Significado
<i>clob, nclob</i>	Objetos de caracteres grandes
<i>rowid</i>	Posición de un registro
<i>blob</i>	Objetos binarios grandes
<i>bfile</i>	Puntero a un fichero binario externo a la base de datos
<i>mlslabel</i>	Utilizado por compatibilidad con el pasado

Notemos, en primer lugar, que los tipos de caracteres tienen dos versiones, y el nombre de una de ellas comienza con *n*. La *n* significa *national*, y los tipos que la indican deben ofrecer soporte para los conjuntos de caracteres nacionales de múltiples bytes: léase japonés, chino y todos esos idiomas que nos suenan a ídem. Además, los tipos *long* y *clob/nclob* se parecen mucho, lo mismo que *long raw* y *blob ...* y es cierto, efectivamente. La diferencia consiste en que los tipos *lob* (de *large objects*) se almacenan más eficientemente y sufren menos restricciones que *long* y *long raw*.

¿Y dónde están nuestros viejos conocidos, los tipos *integer*, *numeric*, *decimal* y *varchar*? Oracle los admite, pero los asocia con alguno de sus tipos nativos. Por ejemplo, *integer* es equivalente a *number(38)*, mientras que *varchar* es lo mismo, hasta la versión 8, que *varchar2*. Sin embargo, Oracle recomienda utilizar siempre *varchar2*, pues en futuras versiones puede variar la semántica de las comparaciones de cadenas de caracteres, para satisfacer el estándar SQL-3.

Estos tipos de datos son los predefinidos por Oracle, y son además los que pueden utilizarse en las definiciones de tablas. Existen los tipos de datos definidos por el usuario, que estudiaremos más adelante, y están los tipos de datos de PL/SQL, que pueden emplearse para definir variables en memoria. Por ejemplo, el tipo *pls_integer* permite definir enteros binarios de 4 bytes, con las operaciones nativas de la CPU de la máquina.

Creación de tablas

Como es de suponer, la sentencia de creación de tablas de Oracle tiene montones de parámetros para configurar el almacenamiento físico de las mismas. Para tener una idea, he aquí una sintaxis abreviada de esta instrucción en Oracle 7 (la versión 8 ha añadido más cláusulas aún):

```
create table [Usuario.] NombreDeTabla (
    DefinicionesDeColumnas&Restricciones
)
[cluster NombreCluster (Columna [, Columna ...])]
[initrans entero] [maxtrans entero]
[pctfree entero] [pctused entero]
```

```
[storage almacenamiento]
[tablespace EspacioDeTabla]
[CláusulaEnable];
```

Gracias a Dios, ocurre lo típico: que podemos olvidarnos de la mayoría de las cláusulas y utilizar valores por omisión. De todos modos, hay un par de opciones de configuración que pueden interesarnos: la especificación de un *espacio de tablas* (*table space*) y el uso de *grupos* o *clusters*. El uso de espacios de tablas es una de las maneras de aprovechar las particiones físicas de una base de datos de Oracle. Los *clusters* de Oracle, por otra parte, permiten ordenar y agrupar físicamente los registros que tienen una misma clave, al estilo de los *clusters* de MS SQL Server. Pero también permiten colocar en una posición cercana los registros de otra tabla relacionados por esa clave. Más adelante dedicaremos una sección a las modalidades de almacenamiento físico de tablas.

La creación de tablas en Oracle, obviando el problema de los parámetros físicos de configuración, no presenta mayores dificultades. Solamente tenga en cuenta los siguientes puntos:

- Recuerde que Oracle traduce los tipos de datos SQL a sus propios tipos nativos. Así que cuando creamos una columna de tipo *integer*, Oracle la interpreta como *number(38)*. C++ Builder entonces se ve obligado a tratarla mediante un componente *TFloatField*. En estos casos, es mejor definir la columna de tipo *number(10)* y activar la opción *ENABLE INTEGERS* en el BDE.
- Las cláusulas **check** no permiten expresiones que hagan referencia a otras tablas. Sí, amigo mío, solamente el humilde InterBase nos ofrece tal potencia.
- A pesar de todo, Oracle 8 no permite la modificación de campos de una restricción de integridad referencial en cascada, aunque sí permite la propagación de borrados mediante la cláusula **on delete cascade**.

Indices en Oracle

Oracle ofrece varias opciones interesantes para la creación de índices. Por ejemplo, permite definir índices con clave invertida. Piense en una tabla en la que se insertan registros, y una de sus columnas es la fecha de inserción. Para el mantenimiento de dicha tabla, los registros se ordenan precisamente por esa columna, y existe la tendencia natural a trabajar más con los últimos registros insertados. Ahora asumamos que en nuestro sistema trabajan concurrentemente un alto número de usuarios. Casi todos estarán manipulando registros dentro del mismo rango de fechas, lo cual quiere decir que en el índice de fechas existirán un par de bloques que estarán siendo constantemente modificados. ¡Tanto disco duro para que al final los usuarios se encaprichen con un par de sectores! Pero como los programadores somos muy listos

(... sí, ese que tiene usted a su lado también ...), hemos creado el índice sobre las fechas del siguiente modo:

```
create index FechaApunte on Apuntes(Fecha) reverse;
```

Cuando usamos la opción **reverse**, las claves se invierten *físicamente*, al nivel de bytes. Como resultado, fechas que antes eran consecutivas se distribuyen ahora de forma aleatoria, y se equilibra la presión sobre el disco duro. Alguna desventaja debe tener este sistema, y es que ahora no se puede aprovechar el índice para recuperar rápidamente los apuntes situados entre tal día y tal otro día.

Oracle permite utilizar las opciones **asc** y **desc** en la creación de índices, por compatibilidad con DB2, pero estas especificaciones son ignoradas. Pruebe, por ejemplo, a crear un índice ascendente y otro descendente para la misma columna o combinación de columnas, y verá cómo Oracle lo rechaza. Sin embargo, y a diferencia de lo que sucede con InterBase, el motor de Oracle puede utilizar el mismo índice para optimizar la ordenación ascendente y descendente en una consulta.

Otra opción curiosa es el uso de índices por mapas de bits (*bitmap indexes*). Tenemos una tabla de clientes, y para cada cliente se almacena la provincia o el estado. Hay 50 provincias en España⁹ y el mismo número de estados en los Estados Unidos. Pero hay un millón de clientes. Con un índice normal, cada clave de este índice debe tener un promedio de 20.000 filas asociadas, listadas de forma secuencial. ¡Demasiado gasto de espacio y tiempo de búsqueda! Un índice por mapas de bits almacena en cada clave una estructura en la que a cada fila corresponde un bit: si está en 0, la fila no pertenece a la clave, si está en 1, sí pertenece.

Hasta la versión 8, las comparaciones entre cadenas de caracteres en Oracle son sensibles a mayúsculas y minúsculas. Lo mismo sucede con los índices.

Organización física de las tablas

Existen varias formas para organizar físicamente los registros de una tabla dentro de una base de datos. ¿Recuerda los *clusters* de MS SQL Server? El recurso de Oracle que más se le parece son las *tablas organizadas por índices* (*index-organized tables*), que se contraponen a la organización tradicional “en montón” (*heap*):

```
create table Detalles (  
    Pedido number(10) not null,  
    Linea number(5) not null,
```

⁹ Si me he equivocado, perdonadme: yo no estudié Geografía Española en la escuela.

```

    -- ... más columnas ...
    primary key (Pedido, Linea)
)
organization index;
```

Como se puede ver en el ejemplo, la cláusula **organization index** se sitúa al finalizar la declaración de la tabla, que debe contener una definición de clave primaria. En tal caso, los registros de la tabla se almacenan en los nodos terminales del índice *b-tree*, ordenados físicamente. Como resultado, la lectura de filas con claves adyacentes es muy eficiente. En la tabla de detalles anterior, por ejemplo, todas las filas de detalles de un pedido estarán situadas una a continuación de la otra.

La organización por índices es una característica de la versión 8 de Oracle. Puede combinarse además con el uso de tablas anidadas, que estudiaremos al final del capítulo. La organización tradicional puede indicarse ahora explícitamente con la cláusula **organization heap**.

Otra alternativa de almacenamiento es el uso de *clusters*, que Oracle interpreta de modo diferente a MS SQL Server. Cabeceras de pedidos y líneas de detalles comparten una misma columna: el número de pedido. Entonces podemos arreglar las cosas para que cada cabecera y sus líneas asociadas se almacenen en la misma página de la base de datos, manteniendo un solo índice sobre el número de pedido para ambas tablas. Este índice puede incluso utilizar la técnica conocida como *hash*, que permite tiempos de acceso muy pequeños. ¿La desventaja? Aunque las búsquedas son muy rápidas, cuesta entonces más trabajo el realizar una inserción, o modificar un número de pedido.

Para definir un *cluster* se utiliza la siguiente instrucción:

```

create cluster cpedidos (
    Numero number(10)
);
```

Note que no se especifican restricciones sobre la columna del *cluster*; esa responsabilidad corresponderá a las tablas que se añadirán más adelante. El siguiente paso es crear un índice para el *cluster*:

```

create index idx_cpedidos on cluster cpedidos;
```

Ya podemos crear tablas para que se almacenen en esta estructura:

```

create table Pedidos(
    Numero number(10) not null primary key,
    -- ... más definiciones ...
)
cluster cpedidos(Numero);
```

```

create table Detalles(
    Pedido number(10) not null,
    Linea number(5) not null,
    -- ... más definiciones ...
)
cluster cpedidos(Pedido);

```

Como podemos ver, no hace falta que coincidan los nombres de columnas en el *cluster* y en las tablas asociadas.

Desafortunadamente, existen limitaciones en el uso de *clusters*. Por ejemplo, no puede dividirse su contenido en particiones y las tablas asociadas no pueden contener columnas de tipo *lob*.

Procedimientos almacenados en PL/SQL

Para crear procedimientos almacenados en Oracle debe utilizar la siguiente instrucción (menciono solamente las opciones básicas):

```

create [or replace] procedure Nombre [(Parámetros)] as
    [Declaraciones]
begin
    Instrucciones
end;
/

```

Los parámetros del procedimiento pueden declararse con los modificadores **in** (se asume por omisión), **out** e **inout**, para indicar el modo de traspaso:

```

create or replace procedure ProximoCodigo(Cod out integer) as
begin
    -- ...
end;
/

```

Si ya conoce InterBase le será fácil iniciarse en PL/SQL, pues la sintaxis en ambos lenguajes es muy parecida, con las siguientes excepciones:

- Las variables locales y parámetros no van precedidas por dos puntos como en InterBase, ni siquiera cuando forman parte de una instrucción SQL.
- La cláusula **into** de una selección se coloca a continuación de la cláusula **select**, no al final de toda la instrucción como en InterBase.
- El operador de asignación en PL/SQL es el mismo de Pascal (**:=**).
- Las instrucciones de control tienen una sintaxis basada en terminadores. Por ejemplo, la instrucción **if** debe terminar con **end if**, **loop**, con **end loop**, y así sucesivamente.

Resulta interesante el uso de procedimientos anónimos en SQL*Plus. Con esta herramienta podemos ejecutar conjuntos de instrucciones arbitrarios, siempre que tengamos la precaución de introducir las declaraciones de variables con la palabra reservada **declare**. Si no necesitamos variables locales, podemos comenzar directamente con la palabra **begin**. Por ejemplo:

```

declare
    Cod integer;
    Cant integer;
begin
    select Codigo into Cod
    from Clientes
    where Nombre = 'Ian Marteens';
    select count(*) into Cant
    from Pedidos
    where RefCliente = Cod;
    if Cant = 0 then
        dbms_output.put_line('Ian Marteens es un tacaño');
    end if;
end;
/

```

Este código asume que hemos instalado el “paquete” *dbms_output*, que nos permite escribir en la salida de SQL*Plus al estilo consola. Hay que ejecutar el siguiente comando de configuración antes de utilizar *put_line*, para que la salida de caracteres se pueda visualizar:

```
set serveroutput on
```

Consultas recursivas

Cuando expliqué los procedimientos almacenados de InterBase, mencioné que una de las justificaciones de este recurso era la posibilidad de poder realizar *clausuras transitivas*, o dicho de forma más práctica, permitir consultas recursivas. En Oracle no hay que llegar a estos extremos, pues ofrece una extensión del comando **select** para plantear directamente este tipo de consultas.

En la base de datos de ejemplo que instala Oracle, hay una tabla que representa los trabajadores de una empresa. La tabla se llama *EMP*, y pertenece al usuario *SCOTT*. Los campos que nos interesan ahora son:

Campo	Significado
<i>EMPNO</i>	Código de este empleado
<i>ENAME</i>	Nombre del empleado
<i>MGR</i>	Código del jefe de este empleado

La parte interesante de esta tabla es que el campo *MGR* hace referencia a los valores almacenados en el campo *EMPNO* de la propia tabla: una situación a todas luces recursiva. ¿Qué tal si listamos todos empleados de la compañía, acompañados por su nivel en la jerarquía? Digamos que el jefe supremo tiene nivel 1, que los que están subordinados directamente al Maharaja tienen nivel 2, y así sucesivamente. En tal caso, la consulta que necesitamos es la siguiente:

```
select level, ename
from   scott.emp
start  with mgr is null
connect by mgr = prior empno
```

Hay dos cláusulas nuevas en la instrucción anterior. Con **start with**, que actúa aproximadamente como una cláusula **where**, indicamos una consulta inicial. En nuestro ejemplo, la consulta inicial contiene una sola fila, la del privilegiado empleado que no tiene jefe. Ahora Oracle realizará repetitivamente la siguiente operación: para cada fila de la última hornada, buscará las nuevas filas que satisfacen la condición expresada en **connect by**. Quiere decir que en este paso se comparan en realidad dos filas, aunque sean de la misma tabla: la fila de la operación anterior y una nueva fila. Es por eso que existe el operador **prior**, para indicar que se trata de una columna de la fila que ya se seleccionó en la última pasada. Es decir, se añaden los empleados cuyos jefes están en el nivel anteriormente explorado. La pseudo columna **level** indica el nivel en el cuál se encuentra la fila activa.

El algoritmo anterior ha sido simplificado conscientemente para poderlo explicar más fácilmente. Pero toda simplificación es una traición a la verdad. En este caso, puede parecer que Oracle utiliza una búsqueda *primero en anchura* para localizar las filas, cuando en realidad la búsqueda se realiza *primero en profundidad*.

Planes de optimización en Oracle

La visualización de los planes de optimización de Oracle es un buen ejemplo de aplicación de las consultas recursivas. Además, le interesará conocer la técnica para poder decidir si Oracle está ejecutando eficientemente sus instrucciones SQL o no.

Para poder averiguar los planes de optimización de Oracle, es necesario que exista una tabla, no importa su nombre, con el siguiente esquema:

```
create table plan_table (
    statement_id          varchar2(30),
    timestamp            date,
    remarks              varchar2(80),
    operation            varchar2(30),
    options              varchar2(30),
    object_node         varchar2(30),
```

```

object_owner          varchar2(30),
object_name           varchar2(30),
object_instance       number,
object_type           varchar2(30),
search_columns        number,
id                    number,
parent_id             number,
position              number,
other                 long
);

```

Ahora se puede utilizar la instrucción **explain plan** para añadir filas a esta tabla. Después necesitaremos visualizar las filas añadidas:

```

explain plan set statement_id = 'ConsultaTonta'
into plan_table for
select * from emp order by empno desc;

```

Observe que se le asocia un identificador literal al plan de ejecución que se va a explicar. Este identificador literal se utiliza en la siguiente consulta, que devuelve el plan generado:

```

select lpad(' ', 2*(level-1)) ||
operation || ' ' ||
options || ' ' ||
object_name || ' ' ||
decode(id, 0, 'Coste=' || position) 'Plan'
from plan_table
start with id = 0 and statement_id = 'ConsultaTonta'
connect by parent_id = prior id and statement_id = 'ConsultaTonta';

```

LPad es una función predefinida de Oracle que sirve para rellenar con espacios en blanco a la izquierda. *Decode* sirve para seleccionar un valor de acuerdo a una expresión.

Cursores

Oracle utiliza cursores para recorrer conjuntos de datos en procedimientos almacenados. Aunque la idea es similar, en general, a los cursores de SQL Server que hemos visto en el capítulo anterior, existen diferencias sintácticas menores. En el capítulo anterior habíamos definido un procedimiento almacenado *ActualizarInventario*, para recorrer todas las filas de un pedido y actualizar las existencias en la tabla *Articulos*. La siguiente es la versión correspondiente en Oracle:

```

create or replace procedure ActualizarInventario(Pedido integer) as
cursor Dets(Ped integer) is
select RefArticulo, Cantidad
from Detalles
where RefPedido = Ped
order by RefArticulo;

```

```

CodArt integer;
Cant integer;

begin
  open Dets(Pedido);
  fetch Dets into CodArt, Cant;
  while Dets%found loop
    update Articulos
    set Pedidos = Pedidos + Cant
    where Codigo = CodArt;
    fetch Dets into CodArt, Cant;
  end loop;
end;
/

```

Esta vez hemos utilizado un cursor con parámetros explícitos. Aunque también se admite que el cursor haga referencia a variables que se encuentran definidas a su alcance, el uso de parámetros hace que los algoritmos queden más claros. Si se definen parámetros, hay que pasar los mismos en el momento de la apertura del cursor con la instrucción **open**. Otra diferencia importante es la forma de detectar el final del cursor. En SQL Server habíamos recurrido a la variable global `@@fetch_status`, mientras que aquí utilizamos el atributo `%found` del cursor. Existen más atributos para los cursores, entre ellos `%notfound` y `%rowcount`; este último devuelve el número de filas recuperadas hasta el momento.

También se puede realizar el recorrido del siguiente modo, aprovechando las instrucciones **loop** y **exit**, para ahorrarnos una llamada a **fetch**:

```

open Dets(Pedido);
loop
  fetch Dets into CodArt, Cant;
  exit when Dets%notfound;
  update Articulos
  set Pedidos = Pedidos + Cant
  where Codigo = CodArt;
end loop;

```

Sin embargo, la forma más clara de plantear el procedimiento aprovecha una variante de la instrucción **for** que es muy similar a la de InterBase:

```

create or replace procedure ActualizarInventario(Pedido integer) as
begin
  for d in (
    select RefArticulo, Cantidad
    from Detalles
    where RefPedido = Ped
    order by RefArticulo) loop
    update Articulos
    set Pedidos = Pedidos + d.Cantidad
    where Codigo = d.RefArticulo;
  end loop;
end;
/

```

Triggers en PL/SQL

Los *triggers* de Oracle combinan el comportamiento de los *triggers* de InterBase y de MS SQL Server, pues pueden dispararse antes y después de la modificación sobre cada fila, o antes y después del procesamiento de un lote completo de modificaciones¹⁰. He aquí una sintaxis abreviada de la instrucción de creación de *triggers*:

```
create [or replace] trigger NombreTrigger
    (before|after) Operaciones on Tabla
    [[referencing Variables] for each row [when (Condicion)]]
declare
    Declaraciones
begin
    Instrucciones
end;
```

Vayamos por partes, comenzando por las *operaciones*. Sucede que un *trigger* puede dispararse para varias operaciones de actualización diferentes:

```
create or replace trigger VerificarReferencia
    before insert or update of RefCliente on Pedidos
    for each row
declare
    i pls_integer;
begin
    select count(*)
    into i
    from Clientes
    where Codigo = :new.RefCliente;
    if i = 0 then
        raise_application_error(-20000,
            'No existe tal cliente');
    end if;
end;
/
```

VerificarReferencia se dispara cuando se inserta un nuevo pedido y cuando se actualiza el campo *RefCliente* de un pedido existente (¿por dónde van los tiros?). Además, la cláusula **for each row** indica que es un *trigger* como Dios manda, que se ejecuta fila por fila. He utilizado también la instrucción *raise_application_error*, para provocar una excepción cuando no exista el cliente referido por el pedido. Observe dos diferencias respecto a InterBase: la variable de correlación *new* necesita ir precedida por dos puntos, y la cláusula **into** se coloca inmediatamente después de la cláusula **select**.

La cláusula **referencing** permite renombrar las variables de correlación, *new* y *old*, en el caso en que haya conflictos con el nombre de alguna tabla:

```
referencing old as viejo new as nuevo
```

¹⁰ Informix es similar en este sentido.

En cuanto a la cláusula **when**, sirve para que el trigger solamente se dispare si se cumple determinada condición. La verificación de la cláusula **when** puede también efectuarse dentro del cuerpo del *trigger*, pero es más eficiente cuando se verifica antes del disparo:

```
create or replace trigger ControlarInflacion
before update on Empleados
for each row when (new.Salario > old.Salario)
begin
    -- Esto es un comentario
end;
/
```

A semejanza de lo que sucede en InterBase, y a diferencia de lo que hay que hacer en MS SQL, Oracle no permite anular explícitamente transacciones dentro del cuerpo de un *trigger*.

La invasión de las tablas mutantes

¿Pondría usted información redundante en una base de datos relacional? Muchos analistas se horrorizarían ante esta posibilidad. Sin embargo, veremos que a veces Oracle no nos deja otra opción. La “culpa” la tiene una regla fundamental que deben cumplir los *triggers* de PL/SQL: no pueden acceder a otras filas de la tabla que se está modificando, excepto a la fila que se modifica, mediante las variables de correlación *new* y *old*. Debo aclarar que esta regla se refiere a *triggers* a nivel de fila.

Supongamos que tenemos un par de tablas para almacenar departamentos y empleados:

```
create table Departamentos (
    ID      integer not null primary key,
    Nombre  varchar(30) not null unique
);

create table Empleados (
    ID      integer not null primary key,
    Nombre  varchar(35) not null,
    Dpto    integer not null references Departamentos(ID)
);
```

Ahora queremos limitar el número de empleados por departamento, digamos que hasta 30. Nuestra primera reacción es crear un *trigger* en el siguiente estilo:

```
create or replace trigger LimitarEmpleados
before insert on Empleados
for each row
declare
    NumeroActual integer;
```

```

begin
    select count(*)
    into NumeroActual
    from Empleados
    where Dpto = :new.Dpto;
    if NumeroActual = 30 then
        raise_application_error(-20001,
            "Demasiados empleados");
    end if;
end;
/

```

Lamentablemente, Oracle no permite este tipo de *trigger*, y lanza un error al intentar ejecutarlo. La dificultad está en la selección de la cantidad, que se realiza accediendo a la misma tabla sobre la cual se dispara el *trigger*. Oracle denomina *tabla mutante* a la tabla que va a ser modificada.

Hay varias soluciones a este problema, pero la más sencilla consiste en mantener en la tabla de departamentos la cantidad de empleados que trabajan en él. Esto implica, por supuesto, introducir una columna redundante:

```

create table Departamentos (
    ID      integer not null primary key,
    Nombre  varchar(30) not null unique,
    Emps    integer default 0 not null
);

```

Debemos actualizar la columna *Emps* cada vez que insertemos un empleado, lo eliminemos o lo cambiemos de departamento. Este, por ejemplo, sería el *trigger* que se dispararía cada vez que insertamos un empleado:

```

create or replace trigger LimitarEmpleados
before insert on Empleados
for each row
declare
    NumeroActual integer;
begin
    select Emps
    into NumeroActual
    from Departamentos
    where ID = :new.Dpto;
    if NumeroActual = 30 then
        raise_application_error(-20001,
            "Demasiados empleados");
    end if;
    update Departamentos
    set Emps = Emps + 1
    where ID = :new.Dpto;
end;
/

```

Esta vez no hay problemas con la tabla mutante, pues la selección se realiza sobre la tabla de departamentos, no sobre los empleados. Aquí también tenemos que tener en

cuenta otra regla de Oracle: en un *trigger* no se puede modificar la clave primaria de una tabla a la cual haga referencia la tabla mutante. *Empleados* hace referencia a *Departamentos* mediante su restricción de integridad referencial. Pero la última instrucción del *trigger* anterior solamente modifica *Emps*, que no es la clave primaria.

Paquetes

En realidad, Oracle siempre ha tratado de adaptarse a las modas. En sus inicios, copió mucho de DB2. En estos momentos, toma prestado del mundo de Java (Oracle8i). Pero a mediados de los 80, el patrón a imitar era el lenguaje Ada. Este lenguaje surgió antes del reciente *boom* de la Programación Orientada a Objetos, pero introdujo algunas construcciones que iban en esa dirección. Una de ellas fueron los *paquetes* o *packages*, que rápidamente fueron asimilados por Oracle.

Un paquete en Oracle contiene una serie de declaraciones de tipos, variables y procedimientos, ofreciendo una forma primitiva de encapsulamiento. Una de las características más importantes de los paquetes es que sus variables públicas conservan sus valores durante todo el tiempo de existencia del paquete. Lo que es más importante ahora para nosotros: cada sesión de Oracle que utiliza un paquete recibe un espacio de memoria separado para estas variables. Por lo tanto, no podemos comunicar información desde una sesión a otra utilizando variables de paquetes. Pero podemos utilizarlas para mantener información de estado durante varias operaciones dentro de una misma sesión.

¿De qué nos sirven los paquetes a nosotros, programadores de C++ Builder? Un paquete no puede utilizarse directamente desde un programa desarrollado con la VCL. Eso sí, puede utilizarse internamente por los *triggers* y procedimientos almacenados de nuestra base de datos. Precisamente, utilizaré un paquete para mostrar cómo se pueden resolver algunas restricciones relacionadas con las tablas mutantes en Oracle.

Vamos a plantearnos la siguiente regla de empresa: un recién llegado a la compañía no puede tener un salario superior a la media. Si intentamos programar la restricción en un *trigger* a nivel de fila fracasaríamos, pues no podríamos consultar la media salarial de la tabla de empleados al ser ésta una tabla mutante durante la inserción. Claro, se nos puede ocurrir almacenar la media salarial en algún registro de otra tabla, pero esto sería demasiado engorroso.

Ya he mencionado que las restricciones correspondientes a tablas mutantes solamente se aplican a los *triggers* a nivel de fila. Un *trigger* a nivel de instrucción sí permite leer valores de la tabla que se va a modificar. El problema es que seguimos necesitando el *trigger* por filas para comprobar la restricción. Si calculamos la media en un

trigger por instrucciones, ¿cómo pasamos el valor calculado al segundo *trigger*? Muy fácil, pues utilizaremos una variable dentro de un paquete.

Primero necesitamos definir la interfaz pública del paquete:

```
create or replace package DatosGlobales as
    MediaSalarial number(15,2) := 0;
    procedure CalcularMedia;
end DatosGlobales;
/
```

Mediante el siguiente *script* suministramos un cuerpo al paquete, en el cual implementaremos el procedimiento *CalcularMedia*:

```
create or replace package body DatosGlobales as
    procedure CalcularMedia is
    begin
        select avg(sal)
        into MediaSalarial
        from emp;
    end;
end DatosGlobales;
/
```

Ahora creamos el *trigger* a nivel de instrucción:

```
create or replace trigger BIEmp
    before insert on Emp
begin
    DatosGlobales.CalcularMedia;
end;
/
```

En este caso tan sencillo, hubiéramos podido ahorrarnos el procedimiento, pero he preferido mostrar cómo se pueden crear. El valor almacenado en la variable del paquete se utiliza en el *trigger* a nivel de fila:

```
create or replace trigger BIEmpRow
    before insert on Emp
    for each row
begin
    if :new.Sal > DatosGlobales.MediaSalarial then
        raise_application_error(-20000,
            '¡Este es un enchufado!');
    end if;
end;
/
```

Actualización de vistas mediante *triggers*

Preste atención a la técnica que describiré en esta sección; cuando estudiemos las actualizaciones en caché del BDE, nos encontraremos con un mecanismo similar, pero que será implementado en el lado cliente de la aplicación. Se trata de permitir actualizaciones sobre vistas que no son actualizables debido a su definición. Consideremos la siguiente vista, que condensa información a partir de la tabla de clientes:

```
create view Ciudades as
  select City, count(CustNo) Cantidad
  from Customer
  group by City;
```

La vista anterior muestra cada ciudad en la cual tenemos al menos un cliente, y la cantidad de clientes existentes en la misma. Evidentemente, la vista no es actualizable, por culpa de la cláusula **group by**. Sin embargo, puede que nos interese permitir ciertas operaciones de modificación sobre la misma. ¿Le doy un motivo? Bien, es muy frecuente que alguien escriba el nombre de una ciudad con faltas de ortografía, y que lo descubramos al navegar por la vista de *Ciudades*. Claro que podemos utilizar una instrucción SQL para corregir todos aquellos nombres de ciudad mal deletreados, pero ¿por qué no simular que actualizamos directamente la vista? Para ello, define el siguiente *trigger*:

```
create trigger CorregirCiudad
  instead of update on Ciudades
  for each row
begin
  update Customer
  set City = :new.City
  where City = :old.City;
end CorregirCiudad;
```

Observe la cláusula **instead of**, que es propia de Oracle. Ahora, la vista permite operaciones de modificación, ya sea desde una aplicación cliente o directamente mediante instrucciones **update**:

```
update Ciudades
set City = 'Munich'
where City = 'München';
```

No he podido resistirme a la tentación de escribir un chiste muy malo de mi cosecha. Supongamos que también definimos el siguiente *trigger*:

```
create trigger EliminarCiudad
  instead of delete on Ciudades
  for each row
begin
  delete from Customer
  where City = :old.City;
end EliminarCiudad;
```

Ahora usted puede montar en cólera divina y lanzar instrucciones como la siguiente:

```
delete from Ciudades
where City in ('Sodoma', 'Gomorra');
```

Secuencias

Las secuencias son un recurso de programación de PL/SQL similar a los generadores de InterBase. Ofrecen un sustituto a las tradicionales tablas de contadores, con la ventaja principal de que la lectura de un valor de la secuencia no impide el acceso concurrente a la misma desde otro proceso. Cuando se utilizan registros con contadores, el acceso al contador impone un bloqueo sobre el mismo que no se libera hasta el fin de la transacción actual. Por lo tanto, el uso de secuencias acelera las operaciones en la base de datos.

Este es un ejemplo básico de definición de secuencias:

```
create sequence CodigoCliente increment by 1 starting with 1;
```

La secuencia anteriormente definida puede utilizarse ahora en un trigger del siguiente modo:

```
create or replace trigger BIClient
before insert on Clientes for each row
begin
  if :new.Codigo is null then
    select CodigoCliente.NextVal
    into  :new.Codigo
    from  Dual;
  end if;
end;
/
```

Aquí hemos utilizado *NextVal*, como si fuera un método aplicado a la secuencia, para obtener un valor y avanzar el contador interno. Si queremos conocer cuál es el valor actual solamente, podemos utilizar el “método” *CurrVal*. Es muy probable que le llame la atención la forma enrevesada que han escogido los diseñadores del lenguaje para obtener el valor de la secuencia. Me explico: *Dual* es una tabla predefinida por Oracle que siempre contiene solamente una fila. Uno pensaría que la siguiente instrucción funcionaría de modo más natural:

```
:new.Codigo := CodigoCliente.NextVal; -- ;;;INCORRECTO!!!
```

Pero no funciona. Sin embargo, sí funcionan instrucciones como la siguiente:

```
insert into UnaTabla(Codigo, Nombre)
values (CodigoCliente.NextVal, 'Your name here')
```

Los mismos problemas que presentan los generadores de InterBase se presentan con las secuencias de Oracle:

- No garantizan la secuencialidad de sus valores, al no bloquearse la secuencia durante una transacción.
- La asignación de la clave primaria en el servidor puede confundir al BDE, cuando se intenta releer un registro recién insertado. La forma correcta de utilizar una secuencia en una aplicación para C++ Builder es escribir un procedimiento almacenado que devuelva el próximo valor de la secuencia, y ejecutar éste desde el evento *OnNewRecord* ó *BeforePost* de la tabla

Tenga en cuenta que el segundo problema que acabamos de explicar se presenta únicamente cuando estamos creando registros y explorando la tabla al mismo tiempo desde un cliente. Si las inserciones transcurren durante un proceso en lote, quizás en el propio servidor, el problema de la actualización de un registro recién insertado no existe.

Como técnica alternativa a las secuencias, cuando deseamos números consecutivos sin saltos entre ellos, podemos utilizar tablas con contadores, al igual que en cualquier otro sistema de bases de datos. Sin embargo, Oracle padece un grave problema: aunque permite transacciones con lecturas repetibles, éstas tienen que ser sólo lectura. ¿Qué consecuencia trae esto? Supongamos que el valor secuencial se determina mediante las dos siguientes instrucciones:

```
select ProximoCodigo
into   :new.Codigo
from   Contadores;
update Contadores
set    ProximoCodigo = ProximoCodigo + 1;
```

Estamos asumiendo que *Contadores* es una tabla con una sola fila, y que el campo *ProximoCodigo* de esa fila contiene el siguiente código a asignar. Por supuesto, este algoritmo no puede efectuarse dentro de una transacción con lecturas repetibles de Oracle. El problema se presenta cuando dos procesos ejecutan este algoritmo simultáneamente. Ambos ejecutan la primera sentencia, y asignan el mismo valor a sus códigos. A continuación, el primer proceso actualiza la tabla y cierra la transacción. Entonces el segundo proceso puede también actualizar y terminar exitosamente, aunque ambos se llevan el mismo valor.

En el capítulo anterior vimos cómo Microsoft SQL Server utilizaba la cláusula **holdlock** en la sentencia de selección para mantener un bloqueo sobre la fila leída hasta el final de la transacción. Oracle ofrece un truco similar, pero necesitamos utilizar un cursor explícito:

```

declare
    cursor Cod is
        select ProximoCodigo
        from Contadores
        for update;
begin
    open Cod;
    fetch Cod into :new.Codigo;
    update Contadores
    set ProximoCodigo = ProximoCodigo + 1
    where current of Cod;
    close Cod;
end;
/

```

Observe la variante de la sentencia **update** que se ejecuta solamente para la fila activa de un cursor.

Tipos de objetos

Ha llegado el esperado momento de ver cómo Oracle mezcla objetos con el modelo relacional. La aventura comienza con la creación de tipos de objetos:

```

create type TClientes as object (
    Nombre          varchar2(35),
    Direccion       varchar2(35),
    Telefono        number(9)
);
/

```

En realidad, he creado un objeto demasiado sencillo, pues solamente posee atributos. Un objeto típico de Oracle puede tener también métodos. Por ejemplo:

```

create type TClientes as object (
    Nombre          varchar2(35),
    Direccion       varchar2(35),
    Telefono        number(9),
    member function Prefijo return varchar2
);
/

```

Como es de suponer, la implementación de la función se realiza más adelante:

```

create or replace type body TClientes as
member function Prefijo return varchar2 is
    C varchar2(9);
begin
    C := to_char(Telefono);
    if substr(C, 1, 2) in ('91', '93') then
        return substr(C, 1, 2);
    end if;
end;

```

```

else
    return substr(C, 1, 3);
end if;
end;
/

```

Como C++ Builder 4 no permite ejecutar, al menos de forma directa, métodos pertenecientes a objetos de Oracle, no voy a insistir mucho sobre el tema. Tenga en cuenta que estos “objetos” tienen una serie de limitaciones:

- No pueden heredar de otras clases.
- No existe forma de esconder los atributos. Aunque definamos métodos de acceso y transformación, de todos modos podremos modificar directamente el valor de los campos del objeto. No obstante, Oracle promete mejoras en próximas versiones.
- No se pueden definir constructores o destructores personalizados.

¿Dónde se pueden utilizar estos objetos? En primer lugar, podemos incrustar columnas de tipo objeto dentro de registros “normales”, o dentro de otros objetos. Suponiendo que existe una clase *TDireccion*, con dos líneas de dirección, código postal y población, podríamos mejorar nuestra definición de clientes de esta forma:

```

create type TClientes as object (
    Nombre          varchar2(35),
    Direccion       TDireccion,
    Telefono         number(9)
);
/

```

Sin embargo, no se me ocurre “incrustar” a un cliente dentro de otro objeto o registro (aunque algunos merecen eso y algo más). Los clientes son objetos con “vida propia”, mientras que la vida de un objeto incrustado está acotada por la del objeto que lo contiene. En compensación, puedo crear una tabla de clientes:

```

create table Clientes of TClientes;

```

En esta tabla de clientes se añaden todas aquellas restricciones que no pudimos establecer durante la definición del tipo:

```

alter table Clientes
    add constraint NombreUnico unique(Nombre);
alter table Clientes
    add constraint ValoresNoNulos check(Nombre <> '');

```

Es posible también especificar las restricciones anteriores durante la creación de la tabla:

```

create table Clientes of TClientes
    Nombre not null,
    check (Nombre <> ''),
    unique (Nombre)
);

```

Muy bien, pero usted estará echando de menos una columna con el código de cliente. Si no, ¿cómo podría un pedido indicar qué cliente lo realizó? ¡Ah, eso es seguir pensando a la antigua! Mire ahora mi definición de la tabla de pedidos:

```

create table Pedidos (
    Numero          number(10) not null primary key,
    Cliente         ref TClientes,
    Fecha           date not null,
    -- ... etcétera ...
);

```

Mis pedidos tienen un campo que es una referencia a un objeto de tipo *TClientes*. Este objeto puede residir en cualquier sitio de la base de datos, pero lo más sensato es que la referencia apunte a una de las filas de la tabla de clientes. La siguiente función obtiene la referencia a un cliente a partir de su nombre:

```

create or replace function RefCliente(N varchar2)
    return ref TClientes as
    Cli ref TClientes;
begin
    select ref(C) into Cli
    from Clientes C
    where Nombre = N;
    return Cli;
end;
/

```

Ahora podemos insertar registros en la tabla de pedidos:

```

insert into Pedidos(Numero, Fecha, Cliente)
values (1, sysdate, RefCliente('Ian Marteens'));

```

La relación que existe entre los pedidos y los clientes es que a cada pedido corresponde a lo máximo un cliente (la referencia puede ser nula, en general). Sin embargo, también es posible representar relaciones uno/muchos: un pedido debe contener varias líneas de detalles. Comenzamos definiendo un tipo para las líneas de detalles:

```

create type TDetalle as object (
    RefArticulo    number(10),
    Cantidad        number(3),
    Precio          number(7,2),
    Descuento       number(3,1),
    member function Subtotal return number
);
/

```

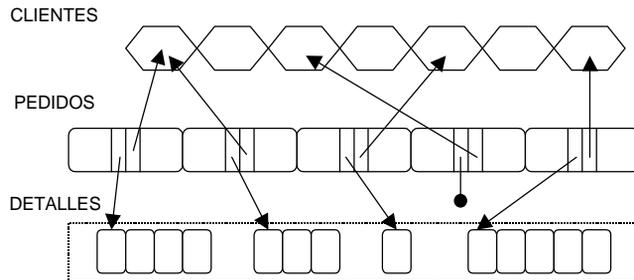
Esta vez no creamos una tabla de objetos, pero en compensación definimos un tipo de tabla anidada:

```
create type TDetalles as table of TDetalle;
```

El nuevo tipo puede utilizarse en la definición de otras tablas:

```
create table Pedidos (
  Numero      number(10) not null primary key,
  Cliente     ref TClientes,
  Fecha       date not null,
  Detalles    TDetalles
)
nested table Detalles store on TablaDetalles;
```

El siguiente diagrama puede ayudarnos a visualizar las relaciones entre los registros de clientes, pedidos y detalles:



Dos pedidos diferentes pueden hacer referencia al mismo cliente. Si eliminamos un cliente al que está apuntando un pedido, Oracle deja una referencia incorrecta, por omisión. Para evitarlo tendríamos que programar *triggers*. Por el contrario, un registro que contiene tablas anidadas es el propietario de estos objetos. Cuando borramos un pedido estamos borrando también todas las líneas de detalles asociadas. En consecuencia, a cada fila de detalles puede apuntar solamente un pedido.

Por último, Oracle 8 permite declarar columnas que sean vectores. Estas columnas se parecen a las tablas anidadas, pero su tamaño máximo está limitado de antemano:

```
create type VDetalles as varray(50) of TDetalle;
```

Aunque he utilizado un tipo de objeto como elemento del vector, también pueden utilizarse tipos simples, como los enteros, las cadenas de caracteres, etc.

DB2 Universal Database

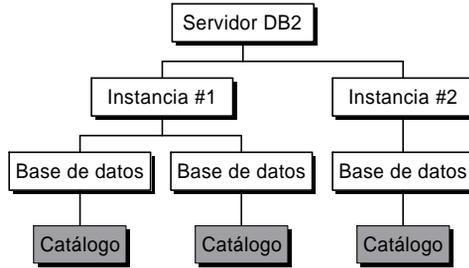
HUBIERA SIDO SUFICIENTE TITULAR ESTE CAPÍTULO con un simple “DB2”. Pero mirándolo bien, eso de *Universal Database* tiene la bastante resonancia acústica como para llamar la atención del lector más despistado. En realidad, DB2 es un sistema que me ha dejado buen sabor de boca, después de las pruebas que he realizado con él, así que no hay nada irónico en el título. Se trata de un sistema con *pedigree*: recuerde que los orígenes de SQL se remontan al *System R*, un prototipo de IBM que nunca llegó a comercializarse. Aunque para ser exactos, UDB (las siglas cariñosas que utilizan los de IBM al referirse a la Base de Datos Universal) está basada en un prototipo más adelantado, que se denominó *System R**.

Una advertencia: el lector notará enseguida las semejanzas entre DB2 y Oracle. No se trata de una coincidencia, pues Oracle ha utilizado como patrón muchos de los avances del *System R* original y del propio DB2, posteriormente. Pero como se trata, en la mayoría de los casos, de aportaciones positivas, no tengo nada que objetar.

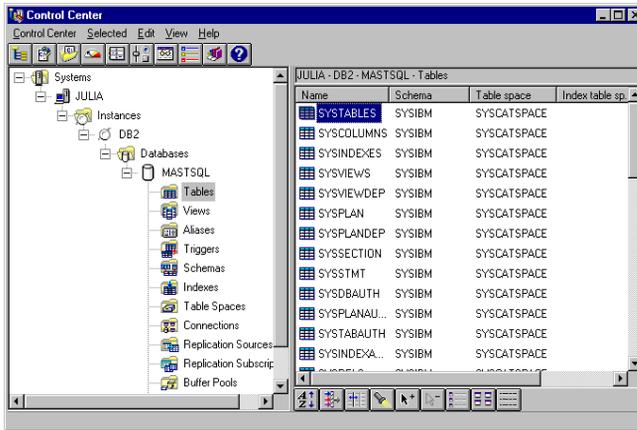
Arquitectura y plataformas

Uno de los principales motivos para la denominación de *Universal Database* es que DB2 puede ejecutarse desde numerosas plataformas, tanto la parte cliente como la servidor. Existen versiones del servidor para Windows NT, OS/2, para varios sabores y colores de UNIX, e incluso para Windows 95. La parte cliente, conocida como *Client Application Enabler*, o *CAE*, puede instalarse también en distintos sistemas operativos. También se soporta un amplio rango de protocolos de comunicación. Es destacable la existencia de una versión “personal” de DB2, que puede ejecutarse en Windows 95.

Un servidor típico permite ejecutar simultáneamente una o más *instancias*, concepto que equivale más o menos al de un espacio de ejecución con nombre. Cada instancia, sin embargo, permite gestionar una o más bases de datos simultáneamente. Cada base de datos tiene su catálogo independiente:



La siguiente imagen muestra en acción a la utilidad *Control Center*, que se puede utilizar como punto de partida para la mayoría de las tareas de administración:



No voy a extenderme sobre las posibilidades de configuración de DB2, pues son muchas. Como cabe esperar de un sistema de gama alta, también se soporta la división de una base de datos en particiones físicas, y se implementan técnicas de replicación. Quiero destacar además que DB2 Enterprise Extended Edition permite el uso de bases de datos distribuidas, y que todas las versiones, excepto la versión personal, vienen preparadas para aprovechar equipos con varios procesadores. Un producto adicional, conocido como DataJoiner, permite extraer datos de otros tipos de servidores, como Oracle, SQL Server, Sybase e Informix, y utilizarlos como si se tratase de información en el formato nativo de DB2.

Aislamiento de transacciones

DB2 implementa todos los niveles de aislamiento de transacciones, y el SQL Link correspondiente del BDE permite trabajar en los modos de lectura confirmada y lecturas repetibles. La implementación de las lecturas repetibles es muy similar a la de MS SQL Server: se colocan bloqueos de lectura sobre los registros leídos durante la

transacción. Una transacción de lectura demasiado larga, en consecuencia, puede bloquear a muchas transacciones de escritura.

Estos son los nombres que DB2 da a los niveles de aislamiento estándar:

Según DB2	Según el estándar
<i>Uncommitted read</i>	<i>Uncommitted read (tiDirtyRead)</i>
<i>Cursor stability</i>	<i>Read Committed</i>
<i>Read stability</i>	<i>Repeatable Read</i>
<i>Repeatable read</i>	<i>Serializable</i>

Recuerde que *serializable* se diferencia de *lecturas repetibles* en que en el último modo pueden todavía aparecer registros fantasmas por culpa de las inserciones.

Una peculiaridad de DB2 consiste en que existe una instrucción para bloquear explícitamente una tabla. Naturalmente, el bloqueo se libera al terminar la transacción actual:

```
lock table NombreTabla in (share|exclusive) mode
```

Tipos de datos

Los tipos de datos básicos de UDB son los siguientes:

Tipo de dato	Significado
<i>smallint</i>	Enteros de 16 bits
<i>integer</i>	Enteros de 32 bits
<i>decimal(p,s)</i>	Equivalente a <i>numeric</i> ; incluye precisión y escala
<i>real, float</i>	Valor flotante de 32 bits
<i>double</i>	Valor flotante de 64 bits
<i>char(n)</i>	Cadenas de longitud fija; $n \leq 254$
<i>varchar(n)</i>	Cadenas de longitud variable; $n \leq 4000$
<i>date</i>	Año, mes y día
<i>time</i>	Hora, minuto y segundo
<i>timestamp</i>	Fecha y hora; precisión en microsegundos
<i>graphic(n)</i>	Bloque binario de longitud fija (≤ 127)
<i>vargraphic(n)</i>	Bloque binario de longitud variable (≤ 2000)

Si no se indica la escala y precisión de un decimal, se asume *decimal(5,0)*. Una importante limitación es que los tipos *varchar* con tamaño mayor de 254 bytes no pueden participar en las cláusulas **order by**, **group by** y **distinct**.

A estos tipos hay que sumarles los tipos *lob* (*large objects*) que se muestran a continuación:

Tipo de dato	Significado
<i>blob</i>	Contiene datos binarios
<i>clob</i>	Contiene caracteres de un solo byte
<i>dbclob</i>	Contiene caracteres Unicode de dos bytes

Los tipos *lob* deben indicar un tamaño máximo de hasta 2GB, utilizando la siguiente sintaxis:

```
Foto          blob(5M) not logged compact,
Documento    clob(500K) logged not compact,
```

Las opción **not compact**, que es la que se asume por omisión, deja espacio libre al final de la zona reservada para el valor, previendo el crecimiento del objeto. Por ejemplo, es poco probable que la foto se sustituya, pero el documento puede ser editado. La otra opción, **not logged**, evita que las operaciones con este campo se guarden en el registro de rehacer transacciones. Así se gana en velocidad, pero si ocurre un fallo, no se pueden recuperar las transacciones ejecutadas sobre la columna desde la última copia de seguridad. Por omisión, se asume **logged**.

En cualquier caso, las modificaciones realizadas a campos *lob* pueden deshacerse sin problemas al anularse una transacción. La opción anterior solamente afecta al registro de rehacer.

Por último, el programador puede crear sus propios tipos de datos:

```
create distinct type Money as decimal(15,2) with comparisons;
```

Cuando creamos un tipo de datos en DB2, los operadores aplicables sobre el tipo base no están automáticamente a disposición del nuevo tipo. Para indicar los operadores que son válidos hay que hacer declaraciones en el siguiente estilo:

```
create function "+"(Money, Money) returns Money
source "+"(Decimal(), Decimal());
create function "-"(Money, Money) returns Money
source "-"(Decimal(), Decimal());
create function sum(Money) returns Money
source sum(Decimal());
```

Aunque hemos utilizado la notación prefijo en la definición de las sumas y restas, podemos sumar dinero en la forma habitual:

```
SalarioBase + PagaExtraordinaria
```

Pero, a no ser que definamos explícitamente la función de multiplicación, no podemos multiplicar dinero por dinero, operación que carece de sentido.

Se pueden añadir funciones adicionales a DB2 programadas externamente. Incluso es posible implementar estas funciones utilizando automatización OLE.

Creación de tablas y restricciones

Todos los objetos que se crean en DB2 se agrupan en *esquemas*, que se crean implícita o explícitamente. Si al crear un objeto no indicamos su esquema, se verifica la existencia de un esquema con el nombre del usuario activo; si no existe, se crea automáticamente dicho esquema. El propietario de todos los esquemas implícitos es el usuario *SYSIBM*. Para crear un esquema explícitamente, y asignarle un propietario que no sea *SYSIBM*, se utiliza la siguiente instrucción:

```
create schema PERSONAL authorization Marteens;
```

A continuación muestro una serie de instrucciones para crear una tabla y colocar sus datos, índices y columnas de gran tamaño en particiones diferentes:

```
create tablespace Datos managed by system
using ('c:\db2data');
create tablespace Indices managed by system
using ('d:\db2idxs');
create long tablespace Blobs managed by database
using (file 'e:\db2blob\blobs.dat' 10000);
create table PERSONAL.Empleados (
Codigo int not null primary key,
Apellidos varchar(30) not null,
Nombre varchar(25) not null,
Foto blob(100K) not logged compact)
in Datos index in Indices long in Blobs;
```

En DB2 podemos utilizar las restricciones habituales en sistemas SQL: claves primarias, claves únicas, integridad referencial y cláusulas **check**. Estas últimas están limitadas a expresiones sobre las columnas de la fila activa, como en Oracle y SQL Server.

Las restricciones de integridad referencial en DB2 admiten las siguientes acciones referenciales:

on delete	on update
<i>cascade</i>	<i>no action</i>
<i>set null</i>	<i>restrict</i>
<i>no action</i>	
<i>restrict</i>	

Es interesante ver cómo DB2 distingue entre **no action** y **restrict**. Por ejemplo, si especificamos **restrict** en la cláusula **on update**, se prohíbe cualquier cambio en la clave primaria de una fila maestra que tenga detalles asociados. Pero si indicamos **no action**, lo único que se exige es que no queden filas de detalles huérfanas. Esta condición puede ser satisfecha si implementamos *triggers* de modificación convenientes sobre la tabla de detalles.

Indices

La sintaxis del comando de creación de índices es:

```
create [unique] index NombreIndice
      on NombreTabla (Columna [asc|desc], ...)
      [include (Columna [asc|desc], ...)]
      [cluster] [pctfree pct]
```

Como podemos ver, se pueden especificar sentidos de ordenación diferente, ascendente o descendente, para cada columna que forma parte del índice. También vemos que se pueden crear índices *agrupados* (*clustered indexes*), como los que ya estudiamos en MS SQL Server.

La cláusula **include** es interesante, pues se utiliza durante la optimización de columnas, y solamente se puede especificar junto a la opción **unique**. Las columnas que se indiquen en **include** se almacenan también en el índice. Por supuesto, la unicidad solamente se verifica con la clave verdadera, pero si el evaluador de consultas necesita también alguna de las columnas de **include** no tiene necesidad de ir a leer su valor al bloque donde está almacenado el registro. Analice la consulta siguiente:

```
select *
from   customer, orders
where  customer.custno = orders.custno and
       customer.vip = 1
```

Si existe un índice único sobre el campo *CustNo* de la tabla *customer*, hemos incluido la columna *vip* en ese índice y el evaluador de consultas decide utilizarlo, nos ahorramos una segunda lectura en la cual, después de seleccionar la clave en el índice, tendríamos que leer el registro para saber si el cliente es una persona muy importante o no.

La longitud total de una clave de índice en DB2 no puede superar los 255 bytes. Aunque no lo mencioné antes, la longitud máxima de un registro es de 4005 bytes, sin contar el espacio ocupado por sus campos de tipo *lob*.

Triggers

DB2 ofrece *triggers* muy potentes. Esta es la sintaxis de creación:

```
create trigger NombreTrigger
  (no cascade before | after)
  (insert | delete | update [of Columnas]) on NombreTabla
  [referencing (old|new|old_table|new_table) as Ident ...]
  [for each (statement | row)] mode db2sql
  [when (Condición)]
  (InstrucciónSimple |
  begin atomic ListaInstrucciones end)
```

La sintaxis y semántica corresponde aproximadamente a la de Oracle, pero existen algunas peculiaridades:

1. Los triggers que se disparan *antes* deben ser siempre a nivel de fila, no de instrucción.
2. Un *trigger* de disparo previo nunca activa a otro trigger de este mismo tipo. Esta característica se indica en la cláusula obligatoria **no cascade before**.
3. En los *triggers* de disparo previo no se pueden ejecutar instrucciones **insert**, **update** o **delete**, aunque se permiten asignaciones a las variables de correlación. Esta es la explicación de la segunda regla.
4. DB2 no permite instrucciones condicionales generales, bucles y otros tipos de extensiones que podemos encontrar en InterBase, PL/SQL y TransactSQL.

Veamos un ejemplo sencillo:

```
create trigger ComprobarAumentoSalarial
  no cascade before update of Salario on Empleados
  referencing old as anterior new as nuevo
  for each row mode db2sql
  when (nuevo.Salario >= 2 * anterior.salario)
  signal sqlstate 70001 ('Aumento desproporcionado');
```

La instrucción **signal** lanza una excepción con dos parámetros. El primero debe ser un código de cinco caracteres, el denominado **sqlstate**, cuyos valores están definidos en el estándar SQL. Para evitar conflictos con valores predefinidos, utilice para el primer carácter un dígito del 7 al 9, o una letra igual o posterior a la 'I'. El mensaje no está limitado a una constante, sino que puede ser cualquier expresión que devuelva una cadena de caracteres.

Al igual que sucede en InterBase y Oracle, una excepción dentro de un *trigger* anula cualquier cambio dentro de la operación activa. Este es el significado de la cláusula **begin atomic**, obligatoria cuando el cuerpo del *trigger* contiene más de una instrucción.

La actual carencia de instrucciones de control tradicionales en DB2 nos obliga a forzar la imaginación para realizar tareas bastante comunes. Es cierto que DB2 suministra extensiones al propio SQL que nos ayudan a superar estas dificultades. Por ejemplo, supongamos que cuando grabamos una cabecera de pedido queremos duplicar en la misma el nombre y la dirección del cliente. Este sería el *trigger* necesario:

```
create trigger DuplicarDatosClientes
no cascade before insert Pedidos
referencing new as nuevo
for each row mode db2sql
when (nuevo.Cliente is not null)
set (nuevo.NombreCliente, nuevo.DirCliente) =
select Nombre, Direccion
from Clientes
where Codigo = nuevo.Cliente;
```

Al no existir una instrucción **if**, estamos obligados a utilizar **when** para ejecutar condicionalmente el *trigger*. DB2 permite definir varios *triggers* para la misma operación sobre la misma tabla, así que tampoco se trata de una limitación radical. Por otra parte, DB2 no reconoce la variante **select/into** de la instrucción de selección, pero vea cómo se sustituye elegantemente con la equivalente instrucción **set**. Dicho sea de paso, tenemos también que recurrir a **set** para cualquier asignación en general:

```
set nuevo.Fecha = current timestamp
```

Consultas recursivas

En el capítulo anterior mostré cómo Oracle permite expresar consultas recursivas. A modo de comparación, veamos como DB2 ofrece un recurso equivalente. Pero comencemos por algo aparentemente más sencillo. Tenemos una tabla con empleados, que almacena también el código del departamento en que estos trabajan. ¿Cuál departamento es el que tiene más empleados? Si existe una tabla de departamentos por separado, en la cual se almacene redundantemente el número de empleados, la respuesta es trivial:

```
select *
from Dept
where NumEmps = (select max(NumEmps) from Dept)
```

Para fastidiar un poco al lector, y obligarlo a acordarse de su SQL, mostraré una consulta equivalente a la anterior:

```
select *
from Dept
where NumEmps >= all (select NumEmps from Dept)
```

Pero, ¿qué pasaría si no existiera el campo redundante *NumEmp* en la tabla de departamentos? Para saber cuántos empleados hay por departamento tendríamos que agrupar las filas de empleados y utilizar funciones de conjunto. Si solamente nos interesa el código de departamento, la consulta sería la siguiente:

```
select DeptNo, count(*)
from Emp
group by DeptNo
```

Y la respuesta a nuestra pregunta sería:

```
select DeptNo, count(*)
from Emp
group by DeptNo
having count(*) >= all (select count(*) from Emp group by DeptNo)
```

Si no se le ha puesto la cara larga después de todo esto, es que usted es un monstruo del SQL. Las cosas se han complicado al tener que evitar calcular el máximo de un total, pues SQL no permite anidar funciones de conjunto. ¿No habrá una forma más sencilla de expresar la pregunta? Sí, si utilizamos una *vista* definida como sigue:

```
create view Plantilla(DeptNo, Total) as
select DeptNo, count(*)
from Emp
group by DeptNo
```

Dada la anterior definición, podemos preguntar ahora:

```
select DeptNo
from Plantilla
where Total = (select max(Total) from Plantilla)
```

Sin embargo, no deja de ser un incordio tener que crear una vista temporal para eliminarla posteriormente. Y esta es precisamente la técnica que nos propone DB2:

```
with Plantilla(DeptNo, Total) as
(select DeptNo, count(*) from Emp group by DeptNo)
select DeptNo
from Plantilla
where Total = (select max(Total) from Plantilla)
```

La instrucción de selección que va entre paréntesis al principio de la consulta, sirve para definir una vista temporal, que utilizamos dos veces dentro de la instrucción que le sigue.

Lo interesante es que **with** puede utilizarse en definiciones recursivas. Supongamos que la tabla de empleados *Emp* contiene el código de empleado (*EmpNo*), el nombre del empleado (*EName*) y el código del jefe (*Mgr*). Si queremos conocer a todos los

empleados que dependen directa o indirectamente de un tal Mr. King, necesitamos la siguiente sentencia:

```
with Empleados(EmpNo, Mgr, EName) as
  ((select EmpNo, Mgr, EName
    from Emp
    where EName = 'KING')
  union all
  (select E1.EmpNo, E1.Mgr, E1.EName
    from Emp E1, Empleados E2
    where E2.EmpNo = E1.Mgr))
select *
from Empleados
```

El truco consiste ahora en utilizar el operador de conjuntos **union all**. El primer operando define la consulta inicial, en la cual obtenemos una sola fila que corresponde al jefe o raíz del árbol de mando. En el segundo operador de la unión realizamos un encuentro entre la tabla de empleados *Emp* y la vista temporal *Empleados*. DB2 interpreta la referencia a la vista temporal como una referencia a los registros generados inductivamente en el último paso. Es decir, en cada nuevo paso se añaden los empleados cuyos jefes se encontraban en el conjunto generado en el paso anterior. El algoritmo se detiene cuando ya no se pueden añadir más empleados al conjunto.

Procedimientos almacenados

Una de las facetas que no me gustan de DB2 es que, aunque podemos crear procedimientos almacenados en el servidor, la implementación de los mismos debe realizarse en algún lenguaje externo: C/C++, Java, etc. En estos procedimientos podemos realizar llamadas directas a la interfaz CLI de DB2, o utilizar sentencias SQL incrustadas que son traducidas por un preprocesador.

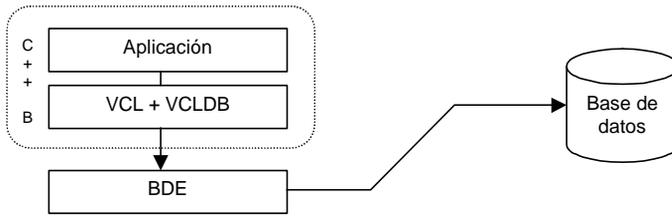
Evidentemente, se produce una ganancia de velocidad en la ejecución de estos procedimientos. ¿Pero a qué precio? En primer lugar, aumenta la complejidad de la programación, pues hay que tener en cuenta, por ejemplo, los convenios de traspaso de parámetros de cada lenguaje. Una modificación en un procedimiento nos obliga a pasar por todo el ciclo de editar/compilar con herramientas externas/instalar, algo que nos evitamos cuando el procedimiento se programa en alguna extensión procedural de SQL. Además, lo típico es que el sistema operativo del servidor sea diferente que el de las estaciones de trabajo. Si el servidor reside en un UNIX o en OS/2, no podemos utilizar Delphi para esta tarea, lo cual nos obliga a utilizar otro lenguaje de programación adicional. En mi humilde opinión, estamos ante una mala decisión de diseño.

El Motor de Datos de Borland

C++ BUILDER ES UN LENGUAJE DE PROPÓSITO GENERAL; nunca me cansaré de repetirlo. Hubo un tiempo en que se puso de moda clasificar los lenguajes por “generaciones”, y mientras más alto el número asignado, supuestamente mejor era el lenguaje. Pascal y C desfilaban en el pelotón de la tercera generación: los lenguajes de “propósito general”, que venían a ser algo así como el lenguaje ensamblador de los de “cuarta generación”. Y a esta categoría “superior” pertenecían los lenguajes de programación para bases de datos que comenzaban a proliferar en aquel entonces.

Algo de razón había, no obstante, para evitar la programación de bases de datos con estos lenguajes de propósito general. Y era la pobre integración de la mayor parte de las bibliotecas de acceso a bases de datos que existían en el mercado. Sin embargo, a partir de entonces los lenguajes tradicionales han experimentado cambios revolucionarios. En la primera parte de este libro hemos mencionado la introducción de la orientación a objetos, el uso de componentes para la comunicación bidireccional y el tratamiento de errores por medio de excepciones. Estos avances han mejorado notablemente la integración y facilidad de uso de bibliotecas creadas por terceros.

C++ Builder y Delphi, en particular, utilizan una arquitectura estructurada en dos niveles. En el nivel inferior se encuentra el *Motor de Datos de Borland*, ó *Borland Database Engine*, más conocido por las siglas BDE, que es un conjunto de funciones agrupadas en bibliotecas dinámicas (DLLs). Esta biblioteca no es orientada a objetos, no permite eventos, y los errores se notifican del modo tradicional: un valor de retorno de la función que falla. Pero el segundo nivel de la arquitectura se encarga de corregir estos “fallos”: el programador de C++ no utiliza directamente las funciones del BDE, sino por mediación de objetos definidos en la VCL, que es la biblioteca de componentes de C++ Builder.



En este capítulo estudiaremos la estructura y filosofía del Motor de Datos, su instalación y configuración. Para terminar, mencionaremos alternativas al uso del BDE.

Qué es, y cómo funciona

Explicábamos en el capítulo anterior las diferencias entre las bases de datos de escritorios y los sistemas SQL. Debido al modelo de comunicación entre las aplicaciones y los datos, una de las implicaciones de estas diferencias es la forma en que se implementa la navegación sobre los datos y, en consecuencia, el estilo de programación que queda determinado. Para las bases de datos de escritorio, por ejemplo, está muy claro el concepto de *posición de registro*. Además, al abrirse una tabla, cuesta prácticamente el mismo tiempo ir al registro 10 que al 10.000. Sin embargo, para las bases de datos SQL la posición de un registro no es una invariante: el orden de inserción es un concepto ajeno a este tipo de formato. Y, posiblemente, buscar el registro número 10.000 de una tabla tarde mil veces más que buscar el décimo registro.

Las operaciones de navegación se implementan muy naturalmente en bases de datos de escritorio. Cuando abrimos una tabla podemos trabajar con sus registros igual que si estuvieran situados en un vector o *array*; es fácil retroceder y avanzar, buscar el primero y el último. En cambio, en una base de datos SQL esta operación puede complicarse. Cuando abrimos una tabla o consulta en estos sistemas normalmente obtenemos un *cursor*, que puede imaginarse un tanto aproximadamente como un fichero de punteros a registros; la implementación verdadera de los cursores depende del sistema SQL concreto y de la petición de apertura realizada. Algunos sistemas, por ejemplo, solamente ofrecen cursores unidireccionales, en los cuales no se permite el retroceso.

En compensación, las bases de datos SQL permiten tratar como tablas físicas al resultado de consultas SQL. Podemos pedir el conjunto de clientes que han comprado los mismos productos que las empresas situadas en Kamchatka, y recibir un cursor del mismo modo que si hubiéramos solicitado directamente la tabla de clientes. Los sistemas basados en registros no han ofrecido, tradicionalmente y hasta fechas recientes, estas facilidades.

Uno de los objetivos fundamentales del diseño del BDE es el de eliminar en lo posible la diferencia entre ambos estilos de programación. Si estamos trabajando con un servidor que no soporta cursores bidireccionales, el Motor de Datos se encarga de implementarlos en el ordenador cliente. De este modo, podemos navegar por cualquier tabla o consulta SQL de forma similar a como lo haríamos sobre una tabla de dBase. Para acercar las bases de datos orientadas a registros a las orientadas a conjuntos, BDE ofrece un intérprete local para el lenguaje SQL.

El Motor de Datos ofrece una arquitectura abierta y modular, en la cual existe un núcleo de funciones implementadas en ficheros DLLs, al cual se le añaden otras bibliotecas dinámicas para acceder a los distintos formatos de bases de datos soportados. El BDE trae ya unos cuantos controladores específicos, aunque recientemente Borland ha puesto a disposición del público el llamado *Driver Development Kit (DDK)* para el desarrollo de nuevos controladores.

Controladores locales y SQL Links

Para trabajar con bases de datos locales, BDE implementa controladores para los siguientes formatos:

- dBase
- Paradox
- Texto ASCII
- FoxPro (a partir de C++ Builder 3)
- Access (a partir de C++ Builder 3)

Los formatos anteriores están disponibles tanto para las versiones Standard, Professional o Client/Server de C++ Builder. Además, se pueden abrir bases de datos de InterBase Local. El servidor local de InterBase viene con las versiones Professional y Client/Server, pero las versiones de 32 bits de este servidor requieren el pago de un pequeño *royalty* para su distribución.

El acceso a bases de datos SQL se logra mediante DLLs adicionales, conocidas como *SQL Links (enlaces SQL)*. Actualmente existen SQL Links para los siguientes formatos:

- InterBase
- Oracle
- Informix
- DB2

- Sybase
- MS SQL Server

Aunque la publicidad es algo confusa al respecto, sin embargo, no basta con instalar los SQL Links sobre un C++ Builder Professional para convertirlo en un C++ Builder Enterprise. Por lo tanto, si aún no ha comprado el producto y tiene en mente trabajar con servidores remotos SQL, vaya directamente a la versión cliente/servidor y no cometa el error de adquirir una versión con menos prestaciones.

Acceso a fuentes de datos ODBC

ODBC (*Open Database Connectivity*) es un estándar desarrollado por Microsoft que ofrece, al igual que el BDE, una interfaz común a los más diversos sistemas de bases de datos. Desde un punto de vista técnico, los controladores ODBC tienen una interfaz de tipo SQL, por lo cual son intrínsecamente inadecuados para trabajar eficientemente con bases de datos orientadas a registros. La especificación también tiene otros defectos, como no garantizar los cursores bidireccionales si el mismo servidor no los proporciona. Sin embargo, por ser una interfaz propuesta por el fabricante del sistema operativo, la mayoría de las compañías que desarrollan productos de bases de datos se han adherido a este estándar.

BDE permite conexiones a controladores ODBC, cuando algún formato no es soportado directamente en el Motor. No obstante, debido a las numerosas capas de software que se interponen entre la aplicación y la base de datos, solamente debemos recurrir a esta opción en casos desesperados. Por suerte, a partir de la versión 4.0 del BDE, se han mejorado un poco los tiempos de acceso vía ODBC.

¿Dónde se instala el BDE?

Para distribuir una aplicación de bases de datos escrita en C++ Builder necesitamos distribuir también el Motor de Datos. En la primera versión de Delphi se incluían los discos de instalación de este producto en un subdirectorio del CD-ROM; solamente había que copiar estos discos y redistribuirlos. A partir de Delphi 2 y del primer C++ Builder no se incluye una instalación prefabricada del BDE, pues siempre podremos generarla mediante InstallShield, que acompaña al producto a partir de la versión profesional.

En dependencia del modelo de aplicación que desarrollemos, la instalación del Motor de Datos tendrá sus características específicas. Los distintos modelos de aplicación son los siguientes:

- *Aplicación monopuesto*: La base de datos y la aplicación residen en el mismo ordenador. En este caso, por supuesto, también debe haber una copia del BDE en la máquina.
- *Aplicación para bases de datos locales en red punto a punto*: Las tablas, normalmente de Paradox ó dBase, residen en un servidor de ficheros. La aplicación se instala en cada punto de la red, o en un directorio común de la red, pero se ejecuta desde cada uno de los puestos. Opcionalmente, la aplicación puede ejecutarse también desde el ordenador que almacena las tablas. En este caso, cada máquina debe ejecutar una copia diferente del BDE. Lo recomendable es instalar el BDE en cada puesto de la red, para evitar problemas con la configuración del Motor de Datos.
- *Aplicaciones cliente/servidor en dos capas*¹¹: Hay un servidor (UNIX, NetWare, WinNT, etcétera) que ejecuta un servidor de bases de datos SQL (InterBase, Oracle, Informix...). Las aplicaciones se ejecutan desde cada puesto de la red y acceden al servidor SQL a través del cliente SQL instalado en el puesto. Nuevamente, el BDE debe ejecutarse desde cada estación de trabajo.
- *Aplicaciones multicapas*: En su variante más sencilla (tres capas), es idéntica a la configuración anterior, pero las estaciones de trabajo no tienen acceso directo al servidor SQL, sino por medio de un servidor de aplicaciones. Este es un ordenador con Windows NT ó Windows 95 instalado, y que ejecuta una aplicación que lee datos del servidor SQL y los ofrece, por medio de comunicación OLEEnterprise, TCP/IP, DCOM o CORBA, a las aplicaciones clientes. En este caso, hace falta una sola copia del BDE, en el servidor de aplicaciones.

El lector se dará cuenta que estas son configuraciones simplificadas, pues una aplicación puede trabajar simultáneamente con más de una base de datos, incluso en diferentes formatos. De este modo, ciertos datos pueden almacenarse en tablas locales para mayor eficiencia, y otros ser extraídos de varios servidores SQL especializados. También, como veremos al estudiar la tecnología Midas, se pueden desarrollar aplicaciones “mixtas”, que combinen accesos en una, dos, tres o más capas simultáneamente.

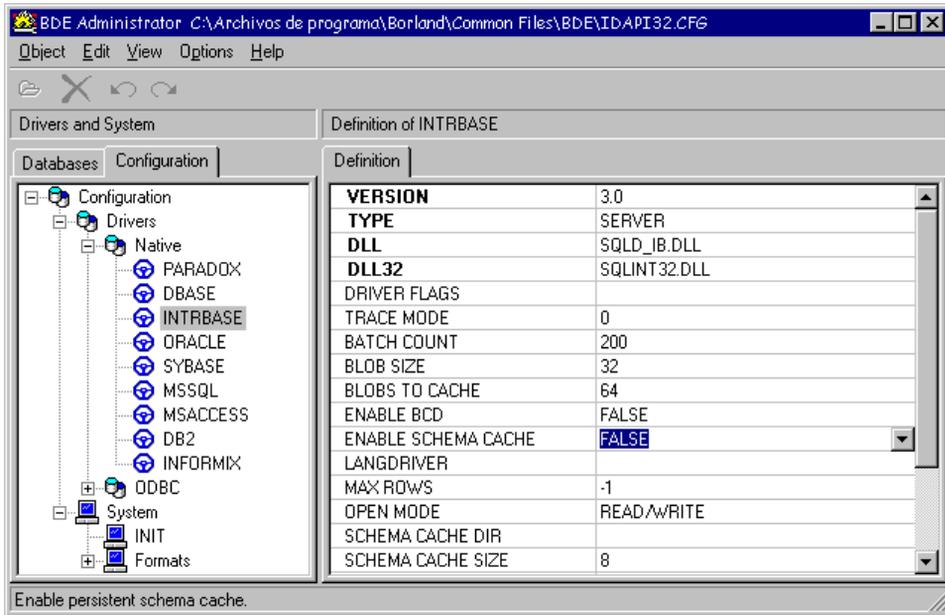
Una de las preguntas más frecuentes acerca del BDE es la posibilidad de instalarlo en un solo puesto de la red, con la doble finalidad de evitar la instalación en cada una de las estaciones de trabajo y las necesarias actualizaciones cada vez que cambie la versión del motor, o alguno de los parámetros de instalación. Bien, esto es técnicamente posible, pero no lo recomiendo. El problema consiste en que es necesario dejar una instalación mínima en los clientes, de todos modos, además que de esta forma se aumenta el tráfico en la red.

¹¹ “Capa” es mi traducción de la palabra inglesa de moda *tier* (no confundir con *tire*). Así, traduciré *multi-tier* como *multicapas*.

El Administrador del Motor de Datos

La configuración del BDE se realiza mediante el programa *BDE Administrator*, que se puede ejecutar por medio del acceso directo existente en el grupo de programas de C++ Builder. Incluso cuando se instala el *runtime* del BDE en una máquina “limpia”, se copia este programa para poder ajustar los parámetros de funcionamiento del Motor.

Existen diferencias entre las implementaciones de este programa para las versiones de 16 y 32 bits. Pero se trata fundamentalmente de diferencias visuales, de la interfaz de usuario, pues los parámetros a configurar y la forma en que están organizados son los mismos. Para simplificar, describiré la interfaz visual del Administrador que viene con las versiones de 32 bits. La siguiente figura muestra el aspecto de este programa:



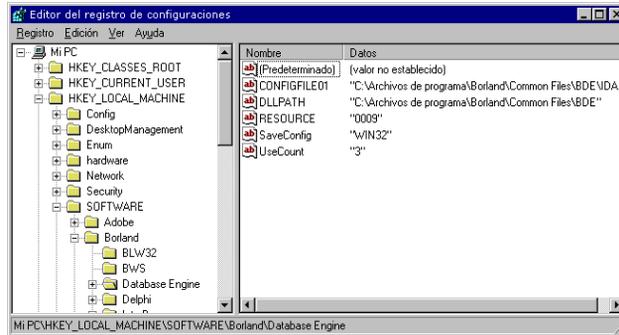
La ventana principal del Administrador está dividida en dos páginas, *Databases* y *Configuration*; comenzaremos por esta última, que permite ajustar los parámetros de los controladores instalados y los parámetros globales de la biblioteca.

Configuración del registro e información de versión

¿Dónde se almacena la información de configuración del BDE? Resulta que se almacena en dos lugares diferentes: la parte principal se guarda en el registro de Windows, mientras que la información de los alias y un par de parámetros especiales va en el

fichero *idapi32.cfg*, cuya ubicación se define en el propio registro. La clave a partir de la cual se encuentran los datos de configuración del BDE es la siguiente:

[HKEY_LOCAL_MACHINE\SOFTWARE\Borland\Database Engine]



Por ejemplo, la cadena *UseCount* indica cuántos productos instalados están utilizando el BDE, mientras que *ConfigFile01* especifica el fichero de configuración a utilizar, y *DLLPath*, el directorio donde están los demás ficheros del BDE.

Para saber la versión del BDE que estamos utilizando, se utiliza el comando de menú *Object | Version information*, que aparece en el Administrador cuando nos encontramos en la página *Databases*:

DLL Name	Version Number	Date	Time	Size (bytes)
IDQBE32.DLL	5.0.0.27 (0)	30/04/98	5:00:00	422400
IDSQL32.DLL	5.0.0.27 (0)	30/04/98	5:00:00	461824
SQLORA32.DLL	5.0.0.27 (0)	30/04/98	5:00:00	414208
SQLORA8.DLL	5.0.0.27 (1)	30/04/98	5:00:00	440832
SQLINT32.DLL	5.0.0.27 (0)	30/04/98	5:00:00	378880
SQLINF32.DLL	5.0.0.27 (0)	30/04/98	5:00:00	394240
SQLDB232.DLL	5.0.0.27 (0)	30/04/98	5:00:00	424960
SQLMSS32.DLL	5.0.0.27 (0)	30/04/98	5:00:00	409088
SQLSYB32.DLL	5.0.0.27 (0)	30/04/98	5:00:00	408576
SQLSSC32.DLL	5.0.0.27 (0)	30/04/98	5:00:00	408064
IDPROV32.DLL	5.0.0.27 (0)	30/04/98	5:00:00	110080
BLW32.DLL	3.0.0.1 (0)	30/04/98	5:00:00	65536
DBCLIENT.DLL	5.0.0.27 (2)	30/04/98	5:00:00	214104

La siguiente tabla resume la historia de las últimas versiones del BDE:

- BDE 4/4.01* Versión original distribuida con Delphi 3/3.01, que incluye por primera vez el acceso directo a Access 95 y a FoxPro.
- BDE 4.5/4.51* Acompaña a Visual dBase 7, e introduce el nuevo formato DBF7 y el acceso directo a Access 97.
- BDE 5/5.01* Acompaña a C++ Builder 4, e introduce el soporte para Oracle 8.

El concepto de alias

Para “aplanar” las diferencias entre tantos formatos diferentes de bases de datos y métodos de acceso, BDE introduce los *alias*. Un alias es, sencillamente, un nombre simbólico para referirnos a un base de datos. Cuando un programa que utiliza el BDE quiere, por ejemplo, abrir una tabla, sólo tiene que especificar un alias y la tabla que quiere abrir. Entonces el Motor de Datos examina su lista de alias y puede saber en qué formato se encuentra la base de datos, y cuál es su ubicación.

Existen dos tipos diferentes de alias: los alias *persistentes* y los alias *locales*, o de *sesión*. Los alias persistentes se crean por lo general con el Administrador del BDE, y pueden utilizarse por cualquier aplicación que se ejecute en la misma máquina. Los alias locales son creados mediante llamadas al BDE realizadas desde una aplicación, y son visibles solamente dentro de la misma. La VCL facilita esta tarea mediante componentes de alto nivel, en concreto mediante la clase *TDatabase*.

Los alias ofrecen a la aplicación que los utiliza independencia con respecto al formato de los datos y su ubicación. Esto vale sobre todo para los alias persistentes, creados con el BDE. Puedo estar desarrollando una aplicación en casa, que trabaje con tablas del alias *datos*. En mi ordenador, este alias está basado en el controlador de Paradox, y las tablas encontrarse en un determinado directorio de mi disco local. El destino final de la aplicación, en cambio, puede ser un ordenador en que el alias *datos* haya sido definido como una base de datos de Oracle, situada en tal servidor y con tal protocolo de conexión. A nivel de aplicación no necesitaremos cambio alguno para que se ejecute en las nuevas condiciones.

A partir de la versión 4.0 del BDE, se introducen los alias *virtuales*, que corresponden a los nombres de fuentes de datos de ODBC, conocidos como DSN (*data source names*). Una aplicación puede utilizar entonces directamente el nombre de un DSN como si fuera un alias nativo del BDE.

Parámetros del sistema

Los parámetros globales del Motor de Datos se cambian en la página *Configuration*, en el nodo *System*. Este nodo tiene a su vez dos subnodos, *INIT* y *Formats*, para los parámetros de funcionamiento y los formatos de visualización por omisión. Aunque estos últimos pueden cambiarse para controlar el formato visual de fechas, horas y valores numéricos, es preferible especificar los formatos desde nuestras aplicaciones, para evitar dependencias con respecto a instalaciones y actualizaciones del BDE. La forma de hacerlo será estudiada en el capítulo 18.

La mayor parte de los parámetros de sistema tienen que ver con el uso de memoria y otros recursos del ordenador. Estos parámetros son:

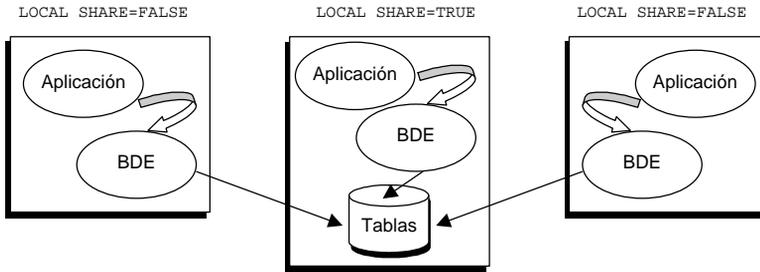
Parámetro	Explicación
<i>MAXFILEHANDLES</i>	Máximo de <i>handles</i> de ficheros admitidos.
<i>MINBUFSIZE</i>	Tamaño mínimo del <i>buffer</i> de datos, en KB.
<i>MAXBUFSIZE</i>	Tamaño máximo del <i>buffer</i> de datos, en KB (múltiplo de 128).
<i>MEMSIZE</i>	Tamaño máximo, en MB, de la memoria consumida por BDE.
<i>LOW MEMORY USAGE LIMIT</i>	Memoria por debajo del primer MB consumida por BDE.
<i>SHAREDMEMSIZE</i>	Tamaño máximo de la memoria para recursos compartidos.
<i>SHAREDMEMLOCATION</i>	Posición en memoria de la zona de recursos compartidos

El área de recursos compartidos se utiliza para que distintas instancias del BDE, dentro de un mismo ordenador, pueden compartir datos entre sí. Las bibliotecas de enlace dinámico de 32 bits tienen un segmento de datos propio para cada instancia, por lo cual tienen que recurrir a ficheros asignados en memoria (*memory mapped files*), un recurso de Windows 95 y NT, para lograr la comunicación entre procesos.

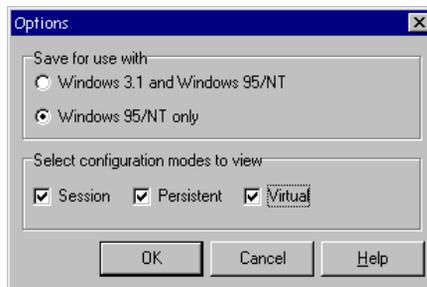
Cuando se va a trabajar con dBase y Paradox es importante configurar correctamente el parámetro *LOCAL SHARE*. Para acelerar las operaciones de bloqueo, BDE normalmente utiliza una estructura en memoria para mantener la lista de bloqueos impuestos en las tablas pertenecientes a los discos locales; esta técnica es más eficiente que depender de los servicios de bloqueo del sistema operativo. Pero solamente vale para aplicaciones que se ejecutan desde un solo puesto. La suposición necesaria para que este esquema funcione es que las tablas sean modificadas siempre por la misma copia del BDE. Si tenemos cinco aplicaciones ejecutándose en un mismo ordenador, todas utilizan la misma imagen en memoria del BDE, y todas utilizan la misma tabla de bloqueos.

Si, por el contrario, las tablas se encuentran en un disco remoto, BDE utiliza los servicios de bloqueo del sistema operativo para garantizar la integridad de los datos. En este caso, el Motor de Datos no tiene acceso a las estructuras internas de otras copias del Motor que se ejecutan en otros nodos de la red. El problema surge cuando dos máquinas, llamémoslas *A* y *B*, acceden a las mismas tablas, que supondremos situadas en *B*. El BDE que se ejecuta en *A* utilizará el sistema operativo para los bloqueos, pero *B* utilizará su propia tabla interna de bloqueos, pues las tablas se encuentran en su disco local. En consecuencia, las actualizaciones concurrentes sobre las tablas destruirán la integridad de las mismas. La solución es cambiar *LOCAL SHARE* a *TRUE* en el ordenador *B*. De cualquier manera, es preferible que en una red punto a punto que ejecute aplicaciones para Paradox y dBase concurrentemente, el servidor de ficheros esté dedicado exclusivamente a este servicio, y no ejecute

aplicaciones en su espacio de memoria; la única razón de peso contra esta política es un presupuesto bajo, que nos obligue a aprovechar también este ordenador.



El otro parámetro importante de la sección *INIT* es *AUTO ODBC*. En versiones anteriores del BDE, este parámetro se utilizaba para crear automáticamente alias en el Motor de Datos que correspondieran a las fuentes de datos ODBC registradas en el ordenador; la operación anterior se efectuaba cada vez que se inicializaba el Motor de Datos. En la versión actual, no se recomienda utilizar esta opción, pues el nuevo modo de configuración virtual de fuentes ODBC la hace innecesaria. Para activar la visualización de alias virtuales, seleccione el comando de menú *Object | Options*, y marque la opción correspondiente en el cuadro de diálogo que aparece a continuación:



Parámetros de los controladores para BD locales

Para configurar controladores de bases de datos locales y SQL, debemos buscar estos controladores en la página *Configuration*, bajo el nodo *Configuration/Drivers/ Native*. Los parámetros para los formatos de bases de datos locales son muy sencillos; comencemos por Paradox.

Quizás el parámetro más importante de Paradox es el directorio del fichero de red: *NET DIR*. Esta variable es indispensable cuando se van a ejecutar aplicaciones con tablas Paradox en una red punto a punto, y debe contener el nombre de un directorio de la red compartido para acceso total. El nombre de este directorio debe escribirse en formato UNC, y debe ser *idéntico* en todos los ordenadores de la red; por ejemplo:

\\SERVIDOR\DirRed

He recalcado el adjetivo “idéntico” porque si el servidor de ficheros contiene una copia del BDE, podemos vernos tentados a configurar *NET DIR* en esa máquina utilizando como raíz de la ruta el nombre del disco local: *c:\DirRed*. Incluso en este ordenador debemos utilizar el nombre UNC. En las versiones de 16 bits del BDE, que no pueden hacer uso de esta notación, se admiten diferencias en la letra de unidad asociada a la conexión de red.

En el directorio de red de Paradox se crea dinámicamente el fichero *pdoxusrs.net*, que contiene los datos de los usuarios conectados a las tablas. Como hay que realizar escrituras en este fichero, se explica la necesidad de dar acceso total al directorio compartido. Algunos administradores inexpertos, en redes Windows 95, utilizan el directorio raíz del servidor de ficheros para este propósito; es un error, porque así estamos permitiendo acceso total al resto del disco.

Cuando se cambia la ubicación del fichero de red de Paradox en una red en la que ya se ha estado ejecutando aplicaciones de bases de datos, pueden producirse problemas por referencias a este fichero almacenadas en ficheros de bloqueos, de extensión *lck*. Mi consejo es borrar todos los ficheros *net* y *lck* de la red antes de modificar el parámetro *NET DIR*.

Los otros dos parámetros importantes de Paradox son *FILL FACTOR* y *BLOCK SIZE*. El primero indica qué porcentaje de un bloque debe estar lleno antes de proceder a utilizar uno nuevo. *BLOCK SIZE* especifica el tamaño de cada bloque en bytes. Paradox permite hasta 65.536 bloques por tabla, por lo que con este parámetro estamos indicando también el tamaño máximo de las tablas. Si utilizamos el valor por omisión, 2.048 bytes, podremos tener tablas de hasta 128MB. Hay que notar que estos parámetros se aplican durante la creación de nuevas tablas; si una tabla existente tiene un tamaño de bloque diferente, el controlador puede utilizarla sin problema alguno.

La configuración de dBase es aún menos problemática. Los únicos parámetros dignos de mención son *MDX BLOCK SIZE* (tamaño de los bloques del índice) y *MEMO FILE BLOCK SIZE* (tamaño de los bloques de los memos). Recuerde que mientras mayor sea el tamaño de bloque de un índice, menor será su profundidad y mejores los tiempos de acceso. Hay que tener cuidado, sin embargo, pues también es más fácil desbordar el *buffer* en memoria, produciéndose más intercambios de páginas.

A partir de la versión 4 del BDE (C++ Builder 3) se ha incluido un controlador para Access. Este controlador actúa como interfaz con el Microsoft Jet Engine, que tiene que estar presente durante el diseño y ejecución de la aplicación. La versión del BDE que acompañaba a C++ Builder 3 solamente permitía utilizar el motor que venía con

Office 95: el DAO 3.0. Si se había instalado Access 97 sobre una instalación previa del 95 no habría problemas, pues la versión anterior se conservaba. No obstante, desde la aparición de la versión 4.5 del BDE, que acompañaba a Visual dBase 7, se ha incluido también el acceso mediante el motor DAO 3.5, que es el que viene con Office 97.

Bloqueos oportunistas

Windows NT permite mejorar la concurrencia en los accesos directos a ficheros de red introduciendo los bloqueos oportunistas. Cuando un cliente de red intenta abrir un fichero situado en el servidor, Windows NT le asigna un bloqueo exclusivo sobre el fichero completo. De esta manera, el cliente puede trabajar eficientemente con copias locales en su caché, realizar escrituras diferidas, etc. La única dificultad consiste en que cuando otro usuario intenta acceder a este mismo fichero, hay que bajarle los humos al oportunista primer usuario, forzándolo a vaciar sus *buffers*.

Bien, resulta que esta maravilla de sistema funciona mal con casi todos los sistemas de bases de datos de escritorio, incluyendo a Paradox. El error se manifiesta cuando las tablas se encuentran en un Windows NT Server, y un usuario encuentra que una tabla completa está bloqueada, cuando en realidad solamente hay un registro bloqueado por otro usuario. Este error es bastante aleatorio: el servidor NT con el que trabajo habitualmente contiene tablas de Paradox para pruebas con miles de registros, y hasta el momento no he tenido problemas. Pero ciertas personas que conozco han pillado este resfriado a la primera.

Para curarnos en salud, lo más sensato es desactivar los bloqueos oportunistas añadiendo una clave al registro de Windows NT Server. El camino a la clave es el siguiente:

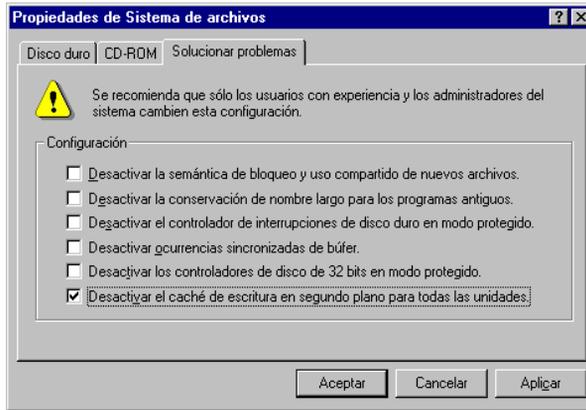
```
[HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\
LanManServer\Parameters]
```

La clave a añadir es la siguiente:

```
EnableOplocks: (DWORD) 00000000
```

También es conveniente modificar las opciones por omisión de la caché de disco en los clientes de Paradox. Por ejemplo, Windows activa por omisión una caché de escritura en segundo plano. Por supuesto, un fallo de la alimentación o un programa colgado pueden afectar a la correcta grabación de las actualizaciones sobre una tabla.

Esta opción puede desactivarse directamente desde la interfaz gráfica de Windows:



Pero si desea que su programa compruebe el estado de la opción e incluso lo modifique, puede mirar la siguiente clave del registro:

```
[HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\FileSystem]
DriveWriteBehind = 00 (DWORD)
```

Por último, muchos programadores recomiendan añadir la siguiente entrada en el registro de Windows 95/98:

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\VXD\VREDIR]
DiscardCacheOnOpen = 01
```

De esta forma, cada vez que se abre una tabla de Paradox se elimina cualquier versión obsoleta de la misma que haya podido quedar en la memoria caché del cliente de red.

Parámetros comunes a los controladores SQL

A pesar de las diferencias entre los servidores SQL disponibles, en la configuración de los controladores correspondientes existen muchos parámetros comunes. La mayor parte de estos parámetros se repiten en la configuración del controlador y en la de los alias correspondientes. El objetivo de esta aparente redundancia es permitir especificar valores por omisión para los alias que se creen posteriormente. Existen, no obstante, parámetros que solamente aparecen en la configuración del controlador, y otros que valen sólo para los alias.

Todos los controladores SQL tienen los siguientes parámetros no modificables:

Parámetro	Significado
<i>VERSION</i>	Número interno de versión del controlador
<i>TYPE</i>	Siempre debe ser <i>SERVER</i>
<i>DLL</i>	DLL de acceso para 16 bits
<i>DLL32</i>	DLL de acceso para 32 bits

En realidad, *DLL* y *DLL32* se suministran para los casos especiales de Informix, Sybase y Oracle, que admiten dos interfaces diferentes con el servidor. Solamente en estos casos deben modificarse los parámetros mencionados.

La siguiente lista muestra los parámetros relacionados con la apertura de bases de datos:

Parámetro	Significado
<i>SERVER NAME</i>	Nombre del servidor
<i>DATABASE NAME</i>	Nombre de la base de datos
<i>USER NAME</i>	Nombre del usuario inicial
<i>OPEN MODE</i>	Modo de apertura: <i>READ/WRITE</i> ó <i>READ ONLY</i>
<i>DRIVER FLAGS</i>	Modifíquelo sólo según instrucciones de Borland
<i>TRACE MODE</i>	Indica a qué operaciones se les sigue la pista

No es necesario, ni frecuente, modificar los cuatro primeros parámetros a nivel del controlador, porque también están disponibles a nivel de conexión a una base de datos específica. En particular, la interpretación del nombre del servidor y del nombre de la base de datos varía de acuerdo al formato de bases de datos al que estemos accediendo. Por ejemplo, el controlador de InterBase utiliza *SERVER NAME*, pero no *DATABASE NAME*.

En los sistemas SQL, la información sobre las columnas de una tabla, sus tipos y los índices definidos sobre la misma se almacena también en tablas de catálogo, dentro de la propia base de datos. Si no hacemos nada para evitarlo, cada vez que nuestra aplicación abra una tabla por primera vez durante su ejecución, estos datos deberán viajar desde el servidor al cliente, haciéndole perder tiempo al usuario y ocupando el ancho de banda de la red. Afortunadamente, es posible mantener una caché en el cliente con estos datos, activando el parámetro lógico *ENABLE SCHEMA CACHE*. Si está activo, se utilizan los valores de los siguientes parámetros relacionados:

Parámetro	Significado
<i>SCHEMA CACHE DIR</i>	El directorio donde se copian los esquemas
<i>SCHEMA CACHE SIZE</i>	Cantidad de tablas cuyos esquemas se almacenan
<i>SCHEMA CACHE TIME</i>	Tiempo en segundos que se mantiene la caché

En el directorio especificado mediante *SCHEMA CACHE DIR* se crea un fichero de nombre *scache.ini*, que apunta a varios ficheros de extensión *scf*, que son los que contienen la información de esquema. Si no se ha indicado un directorio en el parámetro del BDE, se utiliza el directorio de la aplicación. Tenga en cuenta que si la opción está activa, y realizamos cambios en el tipo de datos de una columna de una tabla, tendremos que borrar estos ficheros para que el BDE “note” la diferencia.

Un par de parámetros comunes está relacionado con la gestión de los campos BLOB, es decir, los campos que pueden contener información binaria, como textos grandes e imágenes:

Parámetro	Significado
<i>BLOB SIZE</i>	Tamaño máximo de un BLOB a recibir
<i>BLOBS TO CACHE</i>	Número máximo de BLOBs en caché

Ambos parámetros son aplicables solamente a los BLOBs obtenidos de consultas no actualizables. Incluso en ese caso, parece ser que InterBase es inmune a estas restricciones, por utilizar un mecanismo de traspaso de BLOBs diferente al del resto de las bases de datos.

Por último, tenemos los siguientes parámetros, que rara vez se modifican:

Parámetro	Significado
<i>BATCH COUNT</i>	Registros que se transfieren de una vez con <i>BatchMove</i>
<i>ENABLE BCD</i>	Activa el uso de <i>TBCDField</i> por la VCL
<i>MAX ROWS</i>	Número máximo de filas por consulta
<i>SQLPASSTHRU MODE</i>	Interacción entre SQL explícito e implícito
<i>SQLQRYMODE</i>	Dónde se evalúa una consulta

El BDE, como veremos más adelante, genera implícitamente sentencias SQL cuando se realizan determinadas operaciones sobre tablas. Pero el programador también puede lanzar instrucciones SQL de forma explícita. ¿Pertencen estas operaciones a la misma transacción, o no? El valor por omisión de *SQLPASSTHRU MODE*, que es *SHARED AUTOCOMMIT*, indica que sí, y que cada operación individual de actualización se sitúa automáticamente dentro de su propia transacción, a no ser que el programador inicie explícitamente una.

MAX ROWS, por su parte, se puede utilizar para limitar el número de filas que devuelve una tabla o consulta. Sin embargo, los resultados que he obtenido cuando se alcanza la última fila del cursor no han sido del todo coherentes, por lo que prefiero siempre utilizar mecanismos semánticos para limitar los conjuntos de datos.

Configuración de InterBase

InterBase es el sistema que se configura para el BDE con mayor facilidad. En primer lugar, la instalación del software propio de InterBase en el cliente es elemental: basta con instalar el propio BDE que necesitamos para las aplicaciones en C++ Builder. Sólo necesitamos un pequeño cambio si queremos utilizar TCP/IP como protocolo de comunicación con el servidor: hay que añadir una entrada en el fichero *services*, situado en el directorio de Windows, para asociar un número de puerto al nombre del servicio de InterBase:

```
gds_db          3050/tcp
```

Por supuesto, también podemos tomarnos el trabajo de instalar el software cliente y las utilidades de administración en cada una de las estaciones de trabajo, pero si usted tiene que realizar una instalación para 50 ó 100 puestos no creo que le resulte una opción muy atractiva.

El único parámetro de configuración obligatoria para un alias de InterBase es *SERVER NAME*. A pesar de que el nombre del parámetro se presta a equívocos, lo que realmente debemos indicar en el mismo es la base de datos con la que vamos a trabajar, junto con el nombre del servidor en que se encuentra. InterBase utiliza una sintaxis especial mediante la cual se indica incluso el protocolo de conexión. Por ejemplo, si el protocolo que deseamos utilizar es NetBEUI, un posible nombre de servidor sería:

```
//WILMA/C:/MasterDir/VipInfo.gdb
```

En InterBase, generalmente las bases de datos se almacenan en un único fichero, aunque existe la posibilidad de designar ficheros secundarios para el almacenamiento; es éste fichero el que estamos indicando en el parámetro *SERVER NAME*; observe que, curiosamente, las barras que se emplean para separar las diversas partes de la ruta son las de UNIX. Sin embargo, si en el mismo sistema instalamos el protocolo TCP/IP y lo utilizamos para la comunicación con el servidor, la cadena de conexión se transforma en la siguiente:

```
WILMA:/MasterDir/VipInfo.gdb
```

Como puede comprender el lector, es mejor dejar vacía esta propiedad para el controlador de InterBase, y configurarla solamente a nivel de alias.

Hay algo importante para comprender: los clientes no deben (aunque pueden) tener acceso al directorio donde se encuentre el fichero de la base de datos. Esto es, en el ejemplo anterior *MasterDir* no representa un nombre de recurso compartido, sino un directorio, y preferiblemente un directorio no compartido, por razones de seguridad. La cadena tecleada en *SERVER NAME* es pasada por el software cliente al servicio

instalado en el servidor, y es este programa el que debe tener derecho a trabajar con ese fichero.

Podemos, y debemos, jugar un poco con *DRIVER FLAGS*. Si colocamos el valor 4096 en este parámetro, las grabaciones de registros individuales se producirán más rápidamente, porque el BDE utilizará la función *isc_commit_retaining* para confirmar las transacciones implícitas. Este modo de confirmación graba definitivamente los cambios, pero no crea un nuevo contexto de transacción, sino que vuelve a aprovechar el contexto existente. Esto acelera las operaciones de actualización. Pero lo más importante es que evita que el BDE tenga que releer los cursores activos sobre las tablas afectadas por la transacción. También se puede probar con el valor 512 en *DRIVER FLAGS*, que induce el nivel de aislamiento superior para las transacciones implícitas, el nivel de lecturas repetibles. Si quiere combinar esta constante con la que hemos explicado antes, puede sumarlas:

$$512 + 4096 = 4608$$

El nivel de lecturas repetibles no es el apropiado para todo tipo de aplicaciones. En concreto, las aplicaciones que tienen que estar pendientes de las actualizaciones realizadas en otros puestos no son buenas candidatas a este nivel.

La versión 5.0.1.24 del SQL Link de InterBase añade los siguientes parámetros de configuración:

Parámetro	Significado
<i>COMMIT RETAIN</i>	Evita releer los cursores en transacciones explícitas
<i>WAIT ON LOCKS</i>	Activa el modo de espera por bloqueos en transacciones
<i>ROLE NAME</i>	Permite al usuario asumir un rol inicialmente

El primero de ellos, *COMMIT RETAIN*, surte el mismo que asignar 4096 en el parámetro *DRIVER FLAGS*, pero su acción se centra en las transacciones explícitas. Recuerde que activar esta opción mejorará la eficiencia del BDE al trabajar con transacciones. *WAIT ON LOCKS* fuerza a InterBase a esperar a que un registro esté disponible, en vez de fallar inmediatamente, que era lo que pasaba en versiones anteriores. En el capítulo sobre transacciones ya hemos ponderado las dos alternativas de respuesta a un bloqueo. Por último, *ROLE NAME* permite que el usuario especifique el rol que desea asumir durante la conexión que va a iniciar. Los derechos del rol se suman a los derechos que ya tenía como usuario.

Configuración de MS SQL Server

También es muy sencillo configurar un ordenador para que pueda acceder a un servidor de MS SQL Server. Aunque existe un software para instalar en cada cliente, y que contiene herramientas de administración y consulta, basta con colocar un par de DLLs en el directorio de sistema de Windows. Por ejemplo, si queremos utilizar *named pipes* para la comunicación, necesitamos estas dos DLLs, que podemos extraer del servidor:

<i>ntvdblib.dll</i>	La biblioteca DB-Library de programación.
<i>dbnmpntw.dll</i>	Necesaria para <i>named pipes</i> .

En MS SQL Server, a diferencia de InterBase, *SERVER NAME* representa el nombre del servidor en el cual vamos a situar las bases de datos. Normalmente, este nombre coincide con el nombre del ordenador dentro del dominio, pero puede también ser diferente. Por ejemplo, si el nombre del ordenador contiene acentos o caracteres no válidos para un identificador SQL, la propia instalación del servidor cambia el nombre que identificará al servidor SQL para evitar conflictos. Si vamos a utilizar un único servidor de MS SQL Server, es posible configurar *SERVER NAME* con el nombre del mismo, de modo tal que quede como valor por omisión para cualquier acceso a estas bases de datos. Una vez que hemos especificado con qué servidor lidaremos, hay que especificar el nombre de la base de datos situada en ese servidor, en el parámetro *DATABASE NAME*.

Sin embargo, en contraste con InterBase, el controlador de MS SQL Server tiene muchos parámetros que pueden ser configurados. La siguiente tabla ofrece un resumen de los mismos:

Parámetro	Significado
<i>CONNECT TIMEOUT</i>	Tiempo máximo de espera para una conexión, en segundos
<i>TIMEOUT</i>	Tiempo máximo de espera para un bloqueo
<i>MAX QUERY TIME</i>	Tiempo máximo de espera para la ejecución de una consulta
<i>BLOB EDIT LOGGING</i>	Desactiva las modificaciones transaccionales en campos BLOB
<i>APPLICATION NAME</i>	Identificación de la aplicación en el servidor
<i>HOST NAME</i>	Identificación del cliente en el servidor
<i>DATE MODE</i>	Formato de fecha: 0= <i>mdy</i> , 1= <i>dmy</i> , 2= <i>ymd</i>
<i>TDS PACKET SIZE</i>	Tamaño de los paquetes de intercambio
<i>MAX DBPROCESSES</i>	Número máximo de procesos en el cliente

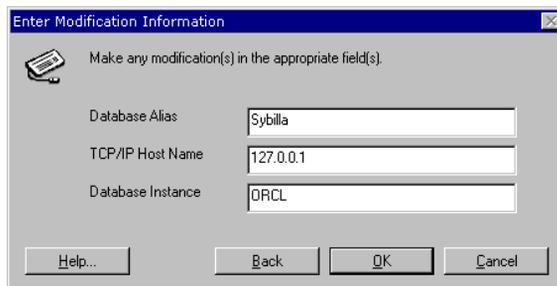
Quizás el parámetro *MAX DBPROCESSES* sea el más importante de todos. La biblioteca de acceso utilizada por el SQL Link de Borland para MS SQL Server es la

DB-Library. Esta biblioteca, en aras de aumentar la velocidad, sacrifica el número de cursores que pueden establecerse por conexión de usuario, permitiendo solamente uno. Así que cada tabla abierta en una estación de trabajo consume una conexión de usuario, que en la versión 6.5 de SQL Server, por ejemplo, necesita 55KB de memoria en el servidor. Es lógico que este recurso se limite a un máximo por cliente, y es ese el valor que se indica en *MAX DBPROCESSES*. Este parámetro solamente puede configurarse en el controlador, no en el alias.

Por su parte, el parámetro *TDS PACKET SIZE* está relacionado con la opción *network packet size* del procedimiento *sp_configure* de la configuración del servidor. Se puede intentar aumentar el valor del parámetro para una mayor velocidad de transmisión de datos, por ejemplo, a 8192. TDS quiere decir *Tabular Data Stream*, y es el formato de transmisión utilizado por MS SQL Server.

Configuración de Oracle

Para configurar un cliente de Oracle, necesitamos instalar obligatoriamente el software SQL Net que viene con este producto, y configurarlo. La siguiente imagen corresponde a uno de los cuadros de diálogo de SQL Net Easy Configuration, la aplicación de configuración que acompaña a Personal Oracle para Windows 95. Los datos corresponden a una conexión que se establece a un servidor situado en la propia máquina. Observe que la dirección IP suministrada es la 127.0.0.1.



La siguiente imagen corresponde al SQL Net Easy Configuration que acompaña a la versión Enterprise 8. A pesar de las diferencias en formato, el procedimiento de conexión sigue siendo básicamente el mismo:



Una vez que hemos configurado un alias con SQL Net, podemos acceder a la base de datos correspondiente. El parámetro *SERVER NAME* del controlador de Oracle se refiere precisamente al nombre de alias que hemos creado con SQL Net.

Parámetro	Significado
<i>VENDOR INIT</i>	Nombre de DLL suministrada por Oracle
<i>NET PROTOCOL</i>	Protocolo de red; casi siempre TNS
<i>ROWSET SIZE</i>	Número de registros que trae cada petición
<i>ENABLE INTEGERS</i>	Traduce columnas de tipo <i>NUMERIC</i> sin escala a campos enteros de la VCL
<i>LIST SYNONYMS</i>	Incluir nombres alternativos para objetos

ROWSET SIZE permite controlar una buena característica de Oracle. Este servidor, al responder a los pedidos de registros por parte de un cliente, se adelanta a nuestras intenciones y envía por omisión los próximos 20 registros del cursor. Así se aprovecha mejor el tamaño de los paquetes de red. Debe experimentar, de acuerdo a su red y a sus aplicaciones, hasta obtener el valor óptimo de este parámetro.

A partir de la versión 5 del BDE, tenemos dos nuevos parámetros para configurar en el controlador de Oracle. El primero es *DLL32*, en el cual podemos asignar uno de los valores *SQLORA32.DLL* ó *SQLORA8.DLL*. Y es que existen dos implementaciones diferentes del SQL Link de Oracle para sus diferentes versiones. El segundo nuevo parámetro es *OBJECT MODE*, que permite activar las extensiones de objetos de Oracle para que puedan ser utilizadas desde C++ Builder.

El problema más frecuente al configurar el SQL Link de Oracle es el mensaje “*Vendor initialization failure*”. Las dos causas más probables: hemos indicado en el parámetro *VENDOR INIT* una DLL que no corresponde a la versión de Oracle instalada, o que dicha DLL no se encuentre en el *PATH* del sistema operativo. La última causa puede parecer una enfermedad infantil fácilmente evitable, pero no lo crea: muchos programas de instalación modifican el *PATH* en el fichero *auto-exec.bat* utilizando nombres largos que contienen espacios. Al arrancar la máqui-

na, el comando da el error “*Demasiados parámetros*”, pues interpreta el espacio como separador de parámetros. Y como casi nadie mira lo que hace su ordenador al arrancar...

Configuración de otros sistemas

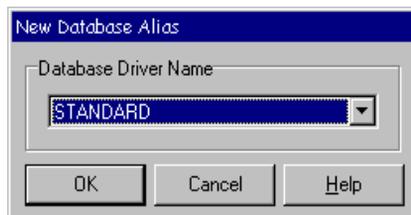
Mencionaré a continuación algunos de los parámetros de configuración de los restantes formatos. Por ejemplo, DB2 utiliza el parámetro *DB2 DSN* para indicar la base de datos con la que se quiere trabajar. Este nombre se crea con la herramienta correspondiente de catalogación en clientes, y es el único parámetro especial del controlador.

Informix tiene un mecanismo similar a MS SQL Server: es necesario indicar el nombre del servidor, *SERVER NAME*, y el de la base de datos, *DATABASE NAME*. El parámetro *LOCK MODE* indica el número de segundos que un proceso espera por la liberación de un bloqueo; por omisión, se espera 5 segundos. El formato de fechas se especifica mediante los parámetros *DATE MODE* y *DATE SEPARATOR*.

Por último, los parámetros de conexión a Sybase son lo suficientemente parecidos a los de MS SQL Server como para no necesitar una discusión adicional.

Creación de alias para bases de datos locales y SQL

Una vez que sabemos cómo manejar los parámetros de configuración de un controlador, es cosa de niños crear alias para ese controlador. La razón es que muchos de los parámetros de los alias coinciden con los de los controladores. El comando de menú *Object|New*, cuando está activa la página *Databases* del Administrador del Motor de Datos, permite crear nuevos alias. Este comando ejecuta un cuadro de diálogo en el que se nos pide el nombre del controlador, y una vez que hemos decidido cuál utilizar, se incluye un nuevo nodo en el árbol, con valores por omisión para el nombre y los parámetros. A continuación, debemos modificar estos valores.



Para crear un alias de Paradox ó dBase debemos utilizar el controlador *STANDARD*. El principal parámetro de este tipo de alias es *PATH*, que debe indicar el directorio (sin la barra final) donde se encuentran las tablas. El parámetro *DEFAULT DRIVER* especifica qué formato debe asumirse si se abre una tabla y no se suministra su extensión, *db* ó *dbf*. Note que este parámetro también existe en la página de configuración del sistema.

Alternativas al Motor de Datos

De cualquier manera, como he explicado al principio del capítulo, BDE no es la única forma de trabajar con bases de datos en C++ Builder, aunque sea la más recomendable. Si no tenemos un SQL Link para nuestra base de datos, y no queremos utilizar una conexión ODBC, contamos con estas opciones:

- Utilizar un sustituto del BDE
- Utilizar componentes derivados directamente de *TDataSet*

La primera opción va quedando obsoleta, pero era la única posibilidad en las versiones 1 y 2 de la VCL. Consistía en reemplazar tanto el motor de datos como las unidades de la VCL que lo utilizaban. Demasiado radical.

La segunda posibilidad aparece con la versión 3 de la VCL, y consiste en desarrollar componentes derivados de la clase *TDataSet*, que es la base de la jerarquía de los objetos de acceso a datos. De esta forma, podemos seguir utilizando el BDE para otros formatos, o no usarlo en absoluto, sin tener que cargar con el código asociado. En realidad existe una tercera alternativa: desarrollar un SQL Link propio con el DDK (*Driver Development Kit*) que ofrece Borland. Pero todavía no conozco ningún producto en este estilo.

Existen, a nivel comercial, alternativas al BDE más o menos exitosas. Por ejemplo, para trabajar con bases de datos de Btrieve, Regatta Systems ofrece Titan, que se comunica directamente con el motor de datos de Btrieve. En realidad, Titan es una *suite* de productos de bases de datos que ofrece también acceso directo a bases de datos de Access y SQL Anywhere. En la misma línea se sitúa Apollo, de SuccessWare, ofreciendo acceso a bases de datos de Clipper, FoxPro y un formato nativo de la compañía. Recuerde que la primera versión de C++ Builder no permitía trabajar con FoxPro ó Access.

2

C++ Builder: navegación y búsquedas

- Conjuntos de datos: tablas
- Acceso a campos
- Validaciones y el Diccionario de Datos
- Controles de datos y fuentes de datos
- Rejillas y barras de navegación
- Índices
- Métodos de búsqueda
- Navegación mediante consultas
- Comunicación cliente/servidor

Parte

Conjuntos de datos: tablas

UN CONJUNTO DE DATOS, para C++ Builder, es cualquier fuente de información estructurada en filas y columnas. Este concepto abarca tanto a las tablas “reales” y las consultas SQL como a ciertos tipos de procedimientos almacenados. Pero también son conjuntos de datos los *conjuntos de datos clientes*, que obtienen su contenido por medio de automatización OLE remota, o a partir de un fichero “plano” local, y que son una de las piezas claves de Midas. Y también lo son las tablas anidadas de Oracle 8, y los conjuntos de datos a la medida que desarrollan otras empresas para acceder a formatos de bases de datos no reconocidos por el Motor de Datos de Borland. Todos estos objetos tienen muchas propiedades, métodos y eventos en común. En este capítulo estudiaremos los conjuntos de datos en general, pero haremos énfasis en las propiedades específicas de las tablas. En capítulos posteriores, nos ocuparemos de las consultas y los procedimientos almacenados.

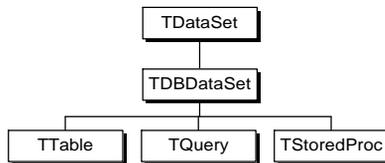
La jerarquía de los conjuntos de datos

La clase *TDataSet* representa una mayor abstracción del concepto de conjunto de datos, sin importar en absoluto su implementación física. Esta clase define características y comportamientos comunes que son heredados por clases especializadas. Estas características son, entre otras:

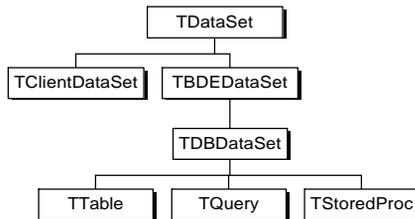
- *Métodos de navegación*: Un conjunto de datos es una colección de registros homogéneos. De estos registros, siempre hay un *registro activo*, que es el único con el que podemos trabajar directamente. Los métodos de navegación permiten gestionar la posición de este registro activo.
- *Acceso a campos*: Todos los registros de un conjunto de datos están estructurados a su vez en campos. Existen mecanismos para descomponer la información almacenada en el registro activo de acuerdo a los campos que forman la estructura del conjunto de datos.
- *Estados del conjunto de datos*: Los conjuntos de datos implementan una propiedad *State*, que los transforman en simples autómatas finitos. Las transiciones entre estados se utilizan para permitir las altas y modificaciones dentro de los conjuntos de datos.

- *Notificaciones a componentes visuales:* Uno de los subsistemas más importantes asociados a los conjuntos de datos envía avisos a todos los componentes que se conectan a los mismos, cada vez que cambia la posición de la fila activa, o cuando se realizan modificaciones en ésta. Gracias a esta técnica es posible asociar controles de edición y visualización directamente a las tablas y consultas.
- *Control de errores:* Cuando detectan un error, los conjuntos de datos disparan eventos que permiten corregir y reintentar la operación, personalizar el mensaje de error o tomar otras medidas apropiadas.

Es interesante ver cómo ha evolucionado la jerarquía de clases a través de la historia de la VCL. En las dos primeras versiones del producto (Delphi 1 y 2, y C++ Builder 1), éstas eran las únicas clases existentes:

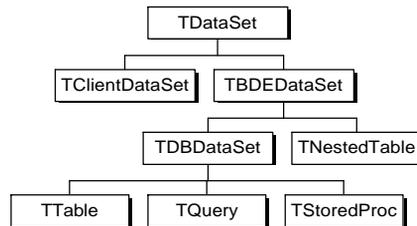


La implementación de la clase *TDataSet*, en aquella época, utilizaba funciones y estructuras de datos del BDE. Además, la relación mutua entre los métodos, eventos y propiedades de esta clase hacían que la implementación de descendientes de la misma fuera bastante fastidiosa. Estos problemas fueron reconocidos durante el desarrollo de la versión 3 de la VCL, que modificó del siguiente modo la jerarquía de clases:



A partir de este momento, *TDataSet* pasó a ser totalmente independiente del BDE. La definición de esta clase reside ahora en la unidad *DB*, mientras que las clases que dependen del BDE (derivadas de *TBDEDataSet*) se han movido a la unidad *DBTables*. Si desarrollamos un programa que no contenga referencias a la unidad *DBTables* (ni a la unidad *BDE* de bajo nivel, por supuesto) no necesitaremos incluir al Motor de Datos en la posterior instalación de la aplicación. Esto es particularmente cierto para las aplicaciones clientes de Midas, que se basan en la clase *TClientDataSet*. Como esta clase desciende directamente de *TDataSet*, no necesita la presencia del BDE para su funcionamiento.

¿Por qué hay dos clases diferentes: *TBDEDataSet* y *TDBDataSet*, si la segunda se deriva directamente de la primera y no tiene hermanos? La clase *TDBDataSet* introduce la propiedad *Database* y otras propiedades y métodos relacionados con la misma. No todo conjunto de datos del BDE tiene que estar asociado a una base de datos. A mí se me ocurre pensar en las tablas en memoria del BDE (no encapsuladas aún en conjuntos de datos). A los desarrolladores de Borland se les ocurrió pensar en *tablas anidadas*, para representar el nuevo tipo de campo de Oracle 8, que puede contener una colección de registros anidados:



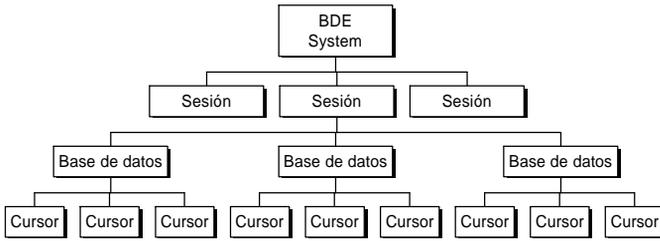
El cambio de arquitectura realizado en la versión 3 ha sido determinante para la creación de sustitutos del BDE, como lo son en este momento Titan (Btrieve y Access), Apollo (Clipper y FoxPro), DOA (Oracle) y otros muchos.

La interacción de los subsistemas de los conjuntos de datos con los demás componentes de la VCL es bastante compleja, y existen muchas dependencias circulares. Para explicar el funcionamiento de los campos, necesitamos saber cómo funcionan las tablas, y viceversa. En este capítulo nos limitaremos al estudio de los métodos de navegación. El capítulo siguiente tratará sobre el acceso a campos. Cuando estudiemos los controles de datos, profundizaremos en las notificaciones a componentes visuales. Por último, al estudiar las actualizaciones veremos en detalle las transiciones de estado y el mecanismo de control de errores.

En vez de comenzar la explicación con la clase abstracta *TDataSet*, lo cual haría imposible mostrar ejemplos, utilizaremos la clase *TTable*, que es además el componente de acceso a datos que encontraremos con mayor frecuencia.

La arquitectura de objetos del Motor de Datos

Cuando utilizamos el BDE para acceder a bases de datos, nuestras peticiones pasan por toda una jerarquía de objetos. El siguiente esquema muestra los tipos de objetos con los que trabaja el BDE, y la relación que existe entre ellos:



El nivel superior se ocupa de la configuración global, inicialización y finalización del *sistema*: el conjunto de instancias del BDE que se ejecutan en una misma máquina. Las *sesiones* representan las diferentes aplicaciones y usuarios que acceden concurrentemente al sistema; en una aplicación de 32 bits pueden existir varias sesiones por aplicación, especialmente si la aplicación soporta concurrencia mediante hilos múltiples.

Cada sesión puede trabajar con varias bases de datos. Estas bases de datos pueden estar en distintos formatos físicos y en diferentes ubicaciones en una red. Su función es controlar la conexión a bases de datos protegidas por contraseñas, la gestión de transacciones y, en general, las operaciones que afectan a varias tablas simultáneamente.

Por último, una vez que hemos accedido a una base de datos, estamos preparados para trabajar con los cursores. Un *cursor* es una colección de registros, de los cuales tenemos acceso a uno solo a la vez, por lo que puede representarse mediante los conjuntos de datos de la VCL. Existen funciones y procedimientos para cambiar la posición del registro activo del cursor, y obtener y modificar los valores asociados a este registro. El concepto de cursor nos permite trabajar con tablas, consultas SQL y con el resultado de ciertos procedimientos almacenados de manera uniforme, ignorando las diferencias entre estas técnicas de obtención de datos.

Cada uno de los tipos de objetos internos del BDE descritos en el párrafo anterior tiene un equivalente directo en la VCL de C++ Builder. La excepción es el nivel principal, el de sistema, algunas de cuyas funciones son asumidas por la clase de sesiones:

Objeto del BDE	Clase de la VCL
Sesiones	<i>TSession</i>
Bases de datos	<i>TDatabase</i>
Cursores	<i>TBDEDataSet</i>

Un programa escrito en C++ Builder no necesita utilizar explícitamente los objetos superiores en la jerarquía a los cursores para acceder a bases de datos. Los componentes de sesión y las bases de datos, por ejemplo, pueden ser creados internamente

por la VCL, aunque el programador puede acceder a los mismos en tiempo de ejecución. Es por esto que podemos postergar el estudio de casi todos estos objetos.

¿Tabla o consulta?

Y bien, usted va a comenzar a desarrollar esa gran aplicación de bases de datos. ¿Qué componente debe utilizar para acceder a sus datos: tablas o consultas? La respuesta depende de qué operaciones va a permitir sobre los datos, del formato y tamaño de los mismos, y de la cantidad de tiempo que quiera invertir en el proyecto. Cuando llegemos al capítulo final de esta parte tendremos elementos suficientes para tomar una decisión, pero podemos adelantar algo ahora.

Si los datos están representados en una base de datos de escritorio, lo más indicado es utilizar las tablas del BDE, mediante el componente *TTable* de la VCL. Este componente accede de forma más o menos directa al fichero donde se almacenan los datos. Los registros se leen por demanda, solamente cuando son necesarios. Si usted tiene una tabla de 100.000 clientes y quiere buscar el último, el BDE realizará un par de operaciones aritméticas y le traerá precisamente el registro deseado. Para las bases de datos de escritorio, el uso de consultas (*TQuery*) es una forma indirecta de acceder a los datos. La ventaja de las consultas es que hay que programar menos, pero lo pagamos en velocidad.

Sin embargo, las consideraciones de eficiencia cambian diametralmente cuando se trata de sistemas cliente/servidor. El enemigo fundamental de estos sistemas es la limitada capacidad de transmisión de datos de las redes actuales. Si usted monta una red local con 100 megabits de tráfico por segundo (tecnología avanzada) esa será la máxima velocidad de transmisión, independientemente de si tiene conectados 10 o 100 ordenadores a la red. Se pueden utilizar trucos: servidores con varios puertos de red, por ejemplo, pero el cuello de botella seguirá localizado en la red.

En tales condiciones, hay operaciones peligrosas para la eficiencia de la red, y la principal de ellas es la navegación indiscriminada. Gumersindo Fernández, un desarrollador de Clipper “de toda la vida” que empieza a trabajar con C++ Builder, ha visto un par de ejemplos en los que se utilizan rejillas de datos. Como Gumersindo sigue al pie de la letra la metodología MMFM¹², pone un par de rejillas en su aplicación para resolver el mantenimiento de sus tablas de 1.000.000 de registros. ¡Y después se queja de que la aplicación va demasiado lenta!

Vaya un momento a un cajero automático, pero no a sacar dinero, sino a observar la interfaz de la aplicación que ejecuta. ¿Ve usted una rejilla por algún lugar? Ciertamente se trata de una interfaz horrible, pero sirve para ilustrar una decisión extrema necesi-

¹² Mientras más fácil, mejor. En inglés sería *KISS: keep it simple, stupid!*

ria para una aplicación cliente/servidor con grandes demandas de tráfico. Resumiendo, en la medida en que pueda evitar operaciones de navegación libre sobre grandes conjuntos de datos, hágalo. ¿Qué operaciones le quedan entonces?

1. Recuperación de pequeños conjuntos de datos (últimos movimientos en la cuenta, por ejemplo).
2. Operaciones de actualización: altas, bajas y modificaciones.

Bien, en los sistemas cliente/servidor estas operaciones se realizan de forma más eficiente utilizando consultas (*TQuery*) y procedimientos almacenados. Un componente *TQuery* puede contener lo mismo una instrucción **select** para recuperar datos que una instrucción **update**, **insert**, **delete** o cualquier otra del lenguaje de definición o control de datos.

¿Qué pasa si no puede evitar navegar de forma libre sobre determinados conjuntos de datos de medianos a grandes? Es aquí donde realmente tendrá que decidir entre tablas y consultas. Aunque deberá esperar hasta el capítulo 25 para conocer los detalles de la implementación de tablas y consultas, he aquí un par de consejos:

1. Las consultas del BDE que contienen un **select** implementan la navegación en dos direcciones a nivel del cliente. Esto quiere decir que para llegar al registro 100.000 deben transferir al cliente los 99.999 registros anteriores. Por lo tanto, descarte las consultas para navegar sobre conjuntos de datos grandes. Por el contrario, al tratarse de un mecanismo relativamente directo de acceder a la interfaz de programación del sistema cliente/servidor, el tiempo de apertura de una consulta es despreciable.
2. El BDE implementa las tablas utilizando internamente instrucciones **select** generadas automáticamente. El objetivo es permitir navegar sobre conjuntos de datos grandes de forma transparente, como si se tratase de una tabla en formato de escritorio. A pesar de la mala fama que tienen entre la comunidad de programadores cliente/servidor, en la mayoría de los casos este objetivo se logra de modo muy eficiente. La mayor desventaja es que el mecanismo de generación automático necesita información sobre el esquema de la tabla, y esta información hay que extraerla desde el servidor durante la apertura de la tabla. En consecuencia, abrir una tabla por primera vez es una operación costosa. No obstante, existen una técnica muy sencilla para disminuir este coste: utilice el parámetro *ENABLE SCHEMA CACHE* para activar la caché de esquemas en el cliente.

Hay que aclarar también que las consultas actualizables necesitan el mismo prólogo de conexión que las tablas. Existe otro factor a tener en cuenta, y es la posibilidad de utilizar actualizaciones en caché para obligar al BDE a realizar las actualizaciones sobre consultas navegables en la forma en que deseemos. Las actualizaciones en caché serán estudiadas en el capítulo 31.

Tablas (por el momento)

Para comenzar el estudio de los conjuntos de datos utilizaremos las tablas: el componente *TTable*. Mediante este componente podemos conectarnos a tablas en cualquiera de los formatos reconocidos por el BDE. El componente *TTable* también permite conectarnos a una *vista* definida en una bases de datos SQL. Las vistas son tablas virtuales, definidas mediante una instrucción **select**, cuyos valores se extraen de otras tablas y vistas. Para más información, puede leer el capítulo 24, que trata sobre las consultas en SQL.

La configuración de una tabla es muy sencilla. Primero hay que asignar el valor de la propiedad *DatabaseName*. En esta propiedad se indica el nombre del alias del BDE donde reside la tabla. Este alias puede ser un alias persistente, como los que creamos con la utilidad de configuración del BDE, o un alias local a la aplicación, creado con el componente *TDatabase*. Esta última técnica se estudia en el capítulo 29, que trata sobre los componentes de bases de datos. Es posible también, si la tabla está en formato Paradox o dBase, asignar el nombre del directorio a esta propiedad. No es, sin embargo, una técnica recomendable pues hace más difícil cambiar dinámicamente la ubicación de las tablas. El siguiente ejemplo muestra cómo asignar un nombre de directorio extraído del registro de Windows a la propiedad *DatabaseName* de una tabla:

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    std::auto_ptr<TRegIniFile> ini(new TRegIniFile(
        "SOFTWARE\\MiEmpresa\\MiAplicacion"));
    Table1->DatabaseName = ini->ReadString("BaseDatos", "Dir", "");
    // ...
    // Código necesario para terminar de configurar la tabla
    // ...
}
```

Una vez que tenemos asignado el nombre del alias o del directorio, podemos especificar el nombre de la tabla dentro de esa base de datos mediante la propiedad *TableName*. Las tablas de Paradox y dBase se almacenan en ficheros, por lo que los nombres de estas tablas llevan casi siempre su extensión, *db* ó *dbf*. Sin embargo, es preferible no utilizar extensiones para este tipo de tablas. La razón es que si no utilizamos extensiones podemos cambiar la base de datos asociada al alias, quizás con la configuración del BDE, y nuestra aplicación podrá trabajar también con tablas en formato SQL. Esta técnica se ilustra en la aplicación *mastapp*, que se instala en el directorio de demostraciones de C++ Builder.



Si no utilizamos extensiones con tablas locales, tenemos una propiedad y un parámetro del BDE para decidir el formato de la tabla. Si la propiedad *TableType* de la tabla vale *ttDefault*, que es el valor por omisión, la decisión se realiza de acuerdo al parámetro *DEFAULT DRIVER* del alias, que casi siempre es *PARADOX*. Pero también podemos asignar *ttDBase*, *ttParadox* ó *ttASCII* a *TableType*, para forzar la interpretación según el formato indicado:

```
Table1->DatabaseName = "BCDEMOS";
// Esta tabla tiene extensión DBF
Table1->TableName = "ANIMALS";
// Sin esta asignación, falla la apertura
Table1->TableType = ttDBase;
```

Para poder extraer, modificar o insertar datos dentro de la tabla, necesitamos que la tabla esté abierta o activa. Esto se controla mediante la propiedad *Active* de la clase. También tenemos los métodos *Open* y *Close*, que realizan asignaciones a *Active*; el uso de estos métodos hace más legible nuestros programas.

En determinados sistemas cliente/servidor, como Oracle y MS SQL Server, los nombres de tablas pueden ir precedidos por el nombre del propietario de la tabla. Por ejemplo, *dbo.customer* significa la tabla *customer* creada por el usuario *dbo*, es decir, el propio creador de la base de datos.

¡No elimine el prefijo de usuario del nombre de la tabla, aunque el truco parezca funcionar! El BDE necesita toda esta información para localizar los índices asociados a la tabla. Sin estos índices, puede que la tabla no pueda actualizarse, o que ocurran problemas (en realidad *bugs*) al cambiar dinámicamente el criterio de ordenación. Esto es especialmente aplicable a MS SQL Server.

Active es una propiedad que está disponible en tiempo de diseño. Esto quiere decir que si le asignamos el valor *True* durante el diseño, la tabla se abrirá automáticamente al cargarse desde el fichero *dfm*. También significa que mientras programamos la aplicación, podemos ver directamente los datos tal y como van a quedar en tiempo de ejecución; esta importante característica no está presente en ciertos sistemas RAD de cuyo nombre no quiero acordarme. No obstante, nunca está de más abrir explícitamente las tablas durante la inicialización del formulario o módulo de datos donde se ha definido. La propiedad *Active* puede, por accidente, quedarse en *False* por culpa de un error en tiempo de diseño, y de este modo garantizamos que en tiempo de ejecución las tablas estén abiertas. Además, aplicar el método *Open* sobre una tabla abierta no tiene efectos negativos, pues la llamada se ignora. En cuanto a cerrar la tabla, el destructor del componente llama automáticamente a *Close*, de modo que durante la destrucción del formulario o módulo donde se encuentra la tabla, ésta se cierra antes de ser destruida.

Un poco antes he mostrado un ejemplo en el que la propiedad *DatabaseName* se asigna en tiempo de ejecución. ¿Cómo hacer entonces para poder visualizar los datos en tiempo de diseño, y no tener que programar a ciegas? Si hacemos *Active* igual a *True* en el formulario, cuando se inicie la aplicación tendremos que cerrar la tabla antes de modificar *DatabaseName*, y el usuario notará el parpadeo del monitor; sin contar que el directorio de pruebas que utilizamos durante el diseño puede no existir en tiempo de ejecución. La solución consiste en realizar el cambio de la propiedad *DatabaseName* durante la respuesta al evento *BeforeOpen*, que se activa justo antes de abrir la tabla. Este evento puede compartirse por todos los componentes cuya base de datos se determine dinámicamente:

```
void __fastcall TForm1::TablasBeforeOpen(TDataSet *DataSet)
{
    auto_ptr<TRegIniFile> ini(new TRegIniFile(
        "SOFTWARE\\MiEmpresa\\MiAplicacion"));
    static_cast<TTable*>(DataSet)->DatabaseName =
        ini->ReadString("BaseDatos", "Dir", "");
}
```

La apertura de la tabla tiene lugar automáticamente después de terminar la ejecución de este método. Observe que el parámetro del evento es del tipo *TDataSet*. Este evento y otros similares serán estudiados en un capítulo posterior.

Exclusividad y bloqueos

La propiedad *Exclusive* permite abrir una tabla en modo exclusivo, garantizando que solamente un usuario esté trabajando con la misma. Por supuesto, la apertura en este modo puede fallar, produciéndose una excepción. *Exclusive*, sin embargo, sólo funciona con Paradox y dBase. Del mismo modo, para estos formatos tenemos los siguientes métodos, que intentan aplicar un bloqueo global sobre la tabla, una vez que está abierta:

```
enum TLockType (ltReadLock, ltWriteLock);

void __fastcall TTable::LockTable(TLockType LockType);
void __fastcall TTable::UnlockTable(TLockType LockType);
```

No cambie el valor de *Exclusive* en tiempo de diseño, pues impediría la apertura de la tabla al depurar el programa. Si desea hacer pruebas en este sentido, debe salir del entorno de desarrollo antes.

Otra propiedad relacionada con el modo de apertura de una tabla es *ReadOnly*, que permite abrir la tabla en el modo sólo lectura.

Conexión con componentes visuales

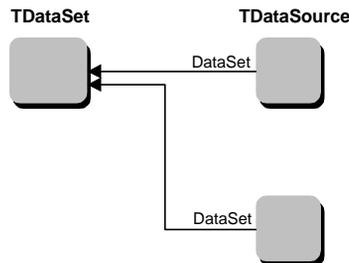
Un conjunto de datos, sea una tabla o una consulta, no puede visualizar directamente los datos con los que trabaja. Para comunicarse con los controles visuales, el conjunto de datos debe tener asociado un componente auxiliar, perteneciente a la clase *TDataSource*. Traduciré esta palabra como *fuentes de datos*, pero trataré de utilizarla lo menos posible, pues el parecido con “conjunto de datos” puede dar lugar a confusiones.

Un objeto *TDataSource* es, en esencia, un “notificador”. Los objetos que se conectan a este componente son avisados de los cambios de estado y de contenido del conjunto de datos controlado por *TDataSource*. Las dos propiedades principales de *TDataSource* son:

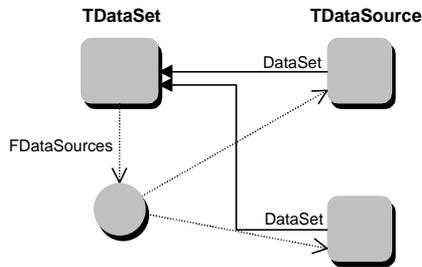
- *DataSet*: Es un puntero, de tipo *TDataSet*, al conjunto de datos que se controla.
- *AutoEdit*: Cuando es *True*, el valor por omisión, permite editar directamente sobre los controles de datos asociados, sin tener que activar explícitamente el modo de edición en la tabla. El modo de edición se explica más adelante.

A la fuente de datos se conectan entonces todos los *controles de datos* que deseemos. Estos controles de datos se encuentran en la página *Data Controls* de la Paleta de Componentes, y todos tienen una propiedad *DataSource* para indicar a qué fuente de datos, y por lo tanto, a qué conjunto de datos indirectamente se conectan. Más adelante, dedicaremos un par de capítulos al estudio de estos controles.

Es posible acoplar más de una fuente de datos a un conjunto de datos. Estas fuentes de datos pueden incluso encontrarse en formularios o módulos de datos diferentes al del conjunto de datos. El propósito de esta técnica es establecer canales de notificación separados. Más adelante, en este mismo capítulo, veremos una aplicación en las relaciones *master/detail*. La técnica de utilizar varias fuentes de datos es posible gracias al mecanismo oculto que emplea C++ Builder. Cuando usted engancha un *data source* a un conjunto de datos esto es lo que ve:

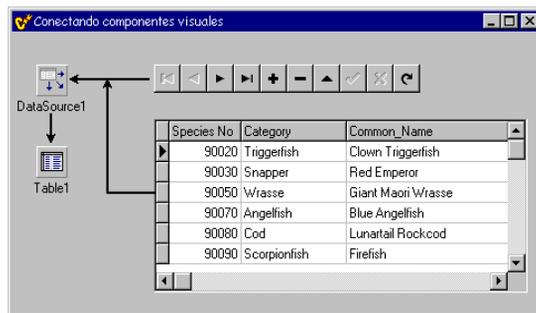


Sin embargo, existe un objeto oculto que pertenece al conjunto de datos, que es una lista de fuentes de datos y que completa el cuadro real:



Un ejemplo común de conexión de componentes es utilizar una *rejilla de datos* (*TDBGrid*) para mostrar el contenido de una tabla o consulta, con la ayuda de una *barra de navegación* (*TDBNavigator*) para desplazarnos por el conjunto de datos y manipular su estado. Esta configuración la utilizaremos bastante en este libro, por lo cual nos adelantamos un poco mostrando cómo implementarla:

Objeto	Propiedad	Valor
<i>Table1: TTable</i>	<i>DatabaseName</i>	El alias de la base de datos
	<i>TableName</i>	El nombre de la tabla
	<i>Active</i>	<i>True</i>
<i>DataSource1: TDataSource</i>	<i>DataSet</i>	<i>Table1</i>
<i>DBGrid1: TDBGrid</i>	<i>DataSource</i>	<i>DataSource1</i>
<i>DBNavigator1: TDBNavigator</i>	<i>DataSource</i>	<i>DataSource1</i>



Es tan frecuente ver a estos componentes juntos, que le recomiendo al lector que guarde la combinación como una plantilla de componentes.

Navegando por las filas

La mayor parte de las operaciones que se realizan sobre un conjunto de datos se aplican sobre la *fila activa* de éste, en particular aquellas operaciones que recuperan o modifican datos. Los valores correspondientes a las columnas de la fila activa se almacenan internamente en un *buffer*, del cual los campos extraen sus valores.

Al abrir un conjunto de datos, inicialmente se activa su primera fila. Qué fila es ésta depende de si el conjunto está ordenado o no. Si se trata de una tabla con un índice activo, o una consulta SQL con una cláusula de ordenación **order by**, el criterio de ordenación es, por supuesto, el indicado. Pero el orden de los registros no queda tan claro cuando abrimos una tabla sin índices activos. Para una tabla SQL el orden de las filas es aún más impredecible que para una tabla de Paradox o dBase. De hecho, el concepto de posición de registro no existe para las bases de datos SQL, debido a la forma en que generalmente se almacenan los registros. Este es el motivo por el cual las barras de desplazamientos vertical de las rejillas de datos de C++ Builder tienen sólo tres posiciones cuando se conectan a un conjunto de datos SQL: al principio de la tabla, al final o en el medio.

Los métodos de movimiento son los siguientes:

Método	Objetivo
<i>First</i>	Ir al primer registro
<i>Prior</i>	Ir al registro anterior
<i>Next</i>	Ir al registro siguiente
<i>Last</i>	Ir al último registro
<i>MoveBy</i>	Moverse la cantidad de filas indicada en el parámetro

Hay dos propiedades que nos avisan cuando hemos llegado a los extremos de la tabla:

Función	Significado
<i>BOF</i>	¿Estamos en el principio de la tabla?
<i>EOF</i>	¿Estamos al final de a tabla?

Combinando estas funciones con los métodos de posicionamiento, podemos crear los siguientes algoritmos de recorrido de tablas:

Hacia adelante	Hacia atrás
<pre>Table1->First(); while (! Table1->Eof) { // Acción Table1->Next(); }</pre>	<pre>Table1->Last(); while (! Table1->Bof) { // Acción Table1->Prior(); }</pre>

La combinación de las propiedades *Bof* y *Eof* nos permite saber si un conjunto de datos está vacío o no. También existe el método *IsEmpty* que realiza esta tarea con más eficiencia:

```
void __fastcall TForm1::acModificacionesClick(TObject *Sender)
{
    acModificaciones->Enabled = ! Table1->IsEmpty();
    // También vale:
    // acModificaciones->Enabled = ! (Table1->Bof && Table1->Eof);
}
```

Si necesitamos conocer la cantidad de filas de un cursor, podemos utilizar el método *RecordCount*. Esta función, sin embargo, debe aplicarse con cautela, pues si estamos tratando con una consulta SQL, su ejecución puede forzar la evaluación completa de la misma, lo cual puede consumir bastante tiempo la primera vez. La evaluación de la propiedad *RecordCount* se realiza en el servidor si el conjunto de datos es un *TTable*, ejecutándose la instrucción SQL siguiente:

```
select count(*)
from   TableName
```

Existe toda una variedad de métodos adicionales para cambiar la fila activa, relacionados con búsquedas por contenido, que se estudiarán en los capítulos que tratan sobre índices y métodos de búsqueda. Por el momento, solamente mencionaré uno más:

```
void __fastcall TTable::GotoCurrent(TTable *OtraTabla);
```

Este método se puede utilizar cuando dos componentes de tablas están trabajando sobre la misma tabla “física”, y deseamos que una de ella tenga la misma fila activa que la otra. Puede que una de estas tablas tenga filtros y rangos activos, mientras que la otra no. Veremos una aplicación de *GotoCurrent* en un capítulo posterior, para realizar búsquedas sobre tablas de detalles.

Marcas de posición

Cuando cambiamos la fila activa de una tabla, es importante saber cómo regresar a nuestro lugar de origen. C++ Builder nos permite recordar una posición para volver más adelante a la misma mediante la técnica de *marcas de posición*. Este es un mecanismo implementado a nivel del BDE. La VCL nos ofrece un tipo de datos, *TBookmark*, que es simplemente un puntero, y tres métodos que lo utilizan:

```
TBookmark __fastcall TDataSet::GetBookmark();
void __fastcall TDataSet::GotoBookmark(TBookmark B);
void __fastcall TDataSet::FreeBookmark(TBookmark B);
```

El algoritmo típico con marcas de posición se muestra a continuación. Tome nota del uso de la instrucción **try/finally** para garantizar el regreso y la destrucción de la marca:

```
// Recordar la posición actual
TBookmark BM = Table1->GetBookmark();
try
{
    // Mover la fila activa ...
}
finally
{
    // Regresar a la posición inicial
    Table1->GotoBookmark(BM);
    // Liberar la memoria ocupada por la marca
    Table1->FreeBookmark(BM);
}
```

En realidad, *TBookmark* es un fósil de la primera versión de la VCL para 16 bits. Para simplificar el trabajo con las marcas de posición puede utilizarse el tipo de datos *TBookmarkStr*, y una nueva propiedad en los conjuntos de datos, *Bookmark*, del tipo anterior. El algoritmo anterior queda de la siguiente forma:

```
// Recordar la posición actual
TBookmarkStr BM = Table1->Bookmark;
try
{
    // Mover la fila activa
}
finally
{
    // Regresar a la posición inicial
    Table1->Bookmark = BM;
}
```

TBookmarkStr está implementada como una cadena de caracteres larga, a la cual se le aplica una conversión de tipos estática. De esta manera, se aprovecha la destrucción automática de la clase para liberar la memoria asociada a la marca, evitándonos el uso de *FreeBookmark*.

Encapsulamiento de la iteración

Como los algoritmos de iteración o recorrido son tan frecuentes en la programación para bases de datos, es conveniente contar con algún tipo de recurso que nos ahorre teclear una y otra vez los detalles repetitivos de esta técnica. Podemos crear un procedimiento que, dado un conjunto de datos en general, lo recorra fila a fila y realice una acción. ¿Qué acción? Evidentemente, cuando programamos este procedimiento no podemos saber cuál; la solución consiste en pasar la acción como parámetro al

procedimiento. El parámetro que indica la acción debe ser un puntero a una función o a un método. Es preferible utilizar un puntero a método, porque de este modo se puede aprovechar el estado del objeto asociado al puntero para controlar mejor las condiciones de recorrido. La declaración de nuestro procedimiento puede ser:

```
void __fastcall RecorrerTabla(TDataSet *DataSet,
    TEventoAccion Action);
```

El tipo del evento, *TEventoAccion*, se debe haber definido antes del siguiente modo:

```
typedef void __fastcall (__closure *TEventoAccion)
    (TDataSet *DataSet, bool &Stop);
```

Recuerde que **__closure** indica que *TEventoAccion* debe apuntar a un método, y no a un procedimiento declarado fuera de una clase.

En el evento pasaremos el conjunto de datos como primer parámetro. El segundo parámetro, de entrada y salida, permitirá que el que utiliza el procedimiento pueda terminar el recorrido antes de alcanzar el fin de tabla, asignando *True* a este parámetro. Finalmente, ésta es la implementación del procedimiento:

```
void __fastcall RecorrerTabla(TDataSet *ADataset,
    TEventoAccion Action)
{
    Screen->Cursor = crHourGlass;
    ADataSet->DisableControls();
    TBookmarkStr BM = ADataSet->Bookmark;
    try
    {
        ADataSet->First();
        bool Stop = False;
        while (!ADataset->Eof && !Stop)
        {
            if (Action)
                Action(ADataset, Stop);
            ADataSet->Next();
        }
    }
    finally
    {
        ADataSet->Bookmark = BM;
        ADataSet->EnableControls();
        Screen->Cursor = crDefault;
    }
}
```

En este procedimiento se ha añadido el cambio de cursor durante la operación. El cursor *crHourGlass* es el famoso reloj de arena que con tanta frecuencia, desgraciadamente, aparece en la pantalla de nuestros ordenadores. Además se han introducido un par de métodos nuevos: *DisableControls* y *EnableControls*. Estos métodos desactivan y reactivan el mecanismo interno de notificación a los controles de datos asociados.

Si estos métodos no se utilizan en la iteración y la tabla tiene controles de datos asociados, cada vez que desplazemos la fila activa los controles se redibujarán. Esto puede ser molesto para el usuario, y es sumamente ineficiente. Hay que garantizar, no obstante, que *EnableControls* vuelva a habilitar las notificaciones, por lo que este método se llama dentro de la cláusula **__finally**.

Una solución intermedia puede ser crear una clase auxiliar, que nos ahorren el prólogo y el epílogo de la iteración. Por ejemplo:

```
class TDataSetIterator
{
private:
    TBookmarkStr BM;
    TDataSet* dataset;
public:
    TDataSetIterator(TDataSet* ds) : dataset(ds)
    {
        Screen->Cursor = crHourGlass;
        ds->DisableControls();
        BM = ds->Bookmark;
        ds->First();
    }
    ~TDataSetIterator()
    {
        dataset->Bookmark = BM;
        dataset->EnableControls();
        Screen->Cursor = crDefault;
    }
};
```

Así nos quedaría ahora un típico bucle de iteración:

```
{
    TDataSetIterator dsi(Table1);

    while (! Table->Eof)
    {
        // Hacer algo con la fila activa ...
        Table1->Next();
    }
}
```

NOTA IMPORTANTE

El hecho de que muchos ejemplos escritos en C++ Builder utilicen métodos de navegación como los anteriores, no le da patente de corso para abusar de los mismos en sus programas, especialmente si está trabajando con una base de datos cliente/servidor. Si usted tiene un bucle de navegación en cuyo interior no existe interacción alguna con el usuario, debe convertirlo lo antes posible en un procedimiento almacenado que se ejecute en el servidor. Así, los registros leídos no circularán por la red, sus programas se ejecutarán más rápido, ahorrarán energía eléctrica y salvarán la selva del Amazonas y la capa de ozono (!!).

Una posible excepción a la nota anterior puede aplicarse cuando los registros se encuentran en la memoria caché de la máquina, si trabajamos con actualizaciones en caché. Y por supuesto, si estamos usando Paradox o dBase, no podemos utilizar procedimientos almacenados. Sin embargo, aquí puede sernos útil desarrollar aplicaciones distribuidas, como veremos más adelante en este libro.

La relación *master/detail*

Es bastante frecuente encontrar tablas dependientes entre sí mediante una relación *uno/muchos*: a una fila de la primera tabla corresponden cero, una o más filas de la segunda tabla. Esta es la relación que existe entre los clientes y sus pedidos, entre las cabeceras de pedidos y sus líneas de detalles, entre Enrique VIII y sus seis esposas... Al definir el esquema de un base de datos, estas relaciones se tienen en cuenta, por ejemplo, para crear restricciones de integridad referencial entre tablas.

La VCL permite establecer un tipo de vínculo entre dos tablas, la relación *master/detail*, con el que podemos representar este tipo de dependencias. En el vínculo intervienen dos tablas, a las que denominaremos la tabla *maestra* y la tabla *dependiente*. La tabla maestra puede ser, en realidad, cualquier tipo de conjunto de datos. La tabla dependiente, en cambio, debe ser un *TTable*. Existe una técnica para hacer que una consulta sea controlada desde otro conjunto de datos, que será estudiada en el capítulo 24.

Cada vez que se cambia la fila activa en la tabla maestra, se restringe el conjunto de trabajo de la tabla dependiente a las filas relacionadas. Si la tabla maestra es la tabla de clientes y la tabla dependiente es la de pedidos, la última tabla debe mostrar en cada momento sólo los pedidos realizados por el cliente activo:

Clientes	
Código	Nombre
1221	Kauai Dive Shoppe
1231	Unisco
1351	Sight Diver

Pedidos		
Número	Cliente	Fecha
1023	1221	2/Jul/88
1076	1221	26/Abr/89
1123	1221	24/Ago/93

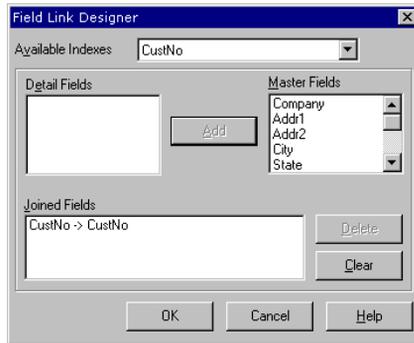
Clientes	
Código	Nombre
1221	Kauai Dive Shoppe
1231	Unisco
1351	Sight Diver

Pedidos		
Número	Cliente	Fecha
1060	1231	1/Mar/89
1073	1231	15/Abr/89
1102	1231	6/Jun/92

Para establecer una relación *master/detail* entre dos tablas solamente hay que hacer cambios en la tabla que va a funcionar como tabla dependiente. Las propiedades de la tabla dependiente que hay que modificar son las siguientes:

Propiedad	Propósito
<i>MasterSource</i>	Apunta a un <i>datasource</i> asociado a la tabla maestra
<i>IndexName</i> ó <i>IndexFieldNames</i>	Criterio de ordenación en la tabla dependiente
<i>MasterFields</i>	Los campos de la tabla maestra que forman la relación

Es necesario configurar una de las propiedades *IndexName* ó *IndexFieldNames*. Aunque estas propiedades se estudiarán en el capítulo sobre índices, nos basta ahora con saber que son modos alternativos de establecer un orden sobre una tabla. Este criterio de ordenación es el que aprovecha C++ Builder para restringir eficientemente el cursor sobre la tabla dependiente. En el ejemplo que mostramos antes, la tabla de pedidos debe estar ordenada por la columna *Cliente*.

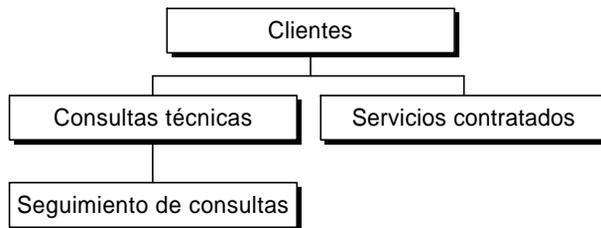


Sin embargo, no tenemos que modificar directamente estas propiedades en tiempo de diseño, pues el editor de la propiedad *MasterFields* se encarga automáticamente de ello. Este editor, conocido como el Editor de Enlaces (*Links Editor*) se ejecuta cuando realizamos una doble pulsación sobre la propiedad *MasterFields*, y su aspecto depende de si la tabla dependiente es una tabla local o SQL. Si la tabla está en formato Paradox o dBase, el diálogo tiene un combo, con la leyenda *AvailableIndexes* para que indiquemos el nombre del índice por el cual se ordena la tabla dependiente. Si la relación *master/detail* está determinada por una restricción de integridad referencial, la mayoría de los sistemas de bases de datos crean de forma automática un índice secundario sobre la columna de la tabla de detalles. Suponiendo que las tablas del ejemplo anterior sean tablas Paradox con integridad referencial definida entre ellas, la tabla de pedidos debe tener un índice secundario, de nombre *Clientes*, que es el que debemos seleccionar. Una vez que se selecciona un índice para la relación, en el cuadro de lista de la izquierda aparecen las columnas pertenecientes al índice elegido. Para este tipo de tablas, el Editor de Enlaces modifica la propiedad *IndexName*: el nombre del índice escogido.

Pero si la tabla pertenece a una base de datos SQL, no aparece la lista de índices, y en el cuadro de lista de la izquierda aparecen todas las columnas de la tabla dependiente. Una tabla SQL no tiene, en principio, limitaciones en cuanto al orden en que se muestran las filas, por lo que basta con especificar las columnas de la tabla dependiente para que la tabla quede ordenada por las mismas. En este caso, la propiedad que modifica el Editor de Enlaces es *IndexFieldNames*: las columnas por las que se ordena la tabla.

En cualquiera de los dos casos, las columnas de la lista de la derecha corresponden a la tabla maestra, y son las que se asignan realmente a la propiedad *MasterFields*. En el ejemplo anterior, *MasterFields* debe tener el valor *Código*.

Gracias a que las relaciones *master/detail* se configuran en la tabla dependiente, y no en la maestra, es fácil crear estructuras complejas basadas en esta relación:

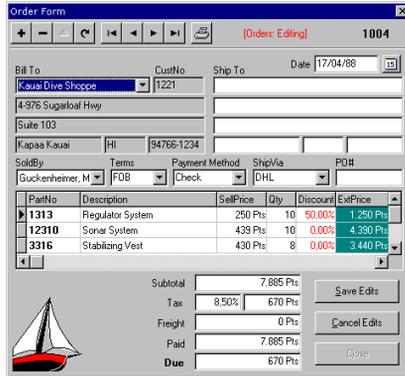


En el diagrama anterior, la tabla de clientes controla un par de tablas dependientes. A su vez, la tabla de consultas técnicas, que depende de la tabla de clientes, controla a la de seguimiento de consultas.

Datos de clientes	
Empresa Anaconda Software S.A.	
Personas de contacto Jan Maiteens	
Friedrich Nietzsche	
Dirección Scotex Valley, 33 Manaos 28100	
Prefijo 666	Teléfono 123-1234
Fax 123-1233	
EMail jmaiteens@anaconda.com	
Alta 23/09/97	
Servicios	
Servicios contratados	
Fecha	Nombre de servicio
23/09/97	Servicios psicológicos
23/09/97	Soporte Técnico Anual Delphi
23/09/97	Soporte Técnico Anual Delphi

La relación *master/detail* no está limitada a representar relaciones uno/muchos, pues también puede utilizarse para la relación inversa. Podemos designar como tabla maestra la tabla de pedidos, y como tabla de detalles la tabla de clientes. En este caso, por cada fila de la primera tabla debe haber exactamente una fila en la tabla de clien-

tes. Si se aprovecha esta relación en una ficha de entrada de pedidos, cuando el usuario introduce un código de cliente en la tabla de pedidos, automáticamente la tabla de clientes cambia su fila activa al cliente cuyo código se ha tecleado. La siguiente imagen, correspondiente a uno de los programas de ejemplo de C++ Builder, muestra esta técnica.



Los cuadros de edición que aparecen con fondo gris en la parte superior izquierda del formulario pertenecen a la tabla de clientes, que está configurada como tabla de detalles de la tabla principal de pedidos. Observe que la rejilla muestra también datos de otra tabla de detalles: las líneas correspondientes al pedido activo.

Otra forma de representar relaciones uno/muchos es mediante las tablas anidadas de Oracle 8. El nuevo componente *TNestedTable* permite visualizar los datos de detalles desde C++ Builder 4. La nueva versión también permite tablas anidadas en conjuntos de datos clientes, que estudiaremos más adelante.

Navegación y relaciones *master/detail*

Spongamos que necesitamos conocer el total facturado por clientes que no viven en los Estados Unidos. Tenemos un formulario con un par de tablas, *tbClientes* y *tbPedidos*, en relación *master/detail*, y queremos aprovechar estos componentes para ejecutar la operación. El país del cliente se almacena en la tabla de clientes, mientras que el total del pedido va en la tabla de pedidos. Y, muy importante para esta sección, las dos tablas están conectadas a sendas rejillas de datos. Estas serán las propiedades de la primera tabla:

	Propiedad	Valor
<i>tbClientes</i>	<i>DatabaseName</i>	<i>bcdemos</i>
	<i>TableName</i>	<i>customer.db</i>
	<i>Active</i>	<i>True</i>

A esta tabla se le asocia un componente *TDataSource*:

	Propiedad	Valor
<i>dsClientes</i>	<i>DataSet</i>	<i>tbClientes</i>

Ahora le toca el turno a la segunda tabla:

	Propiedad	Valor
<i>tbPedidos</i>	<i>DatabaseName</i>	<i>bcdemos</i>
	<i>TableName</i>	<i>orders.db</i>
	<i>MasterSource</i>	<i>dsClientes</i>
	<i>IndexName</i>	<i>CustNo</i>
	<i>MasterFields</i>	<i>CustNo</i>
	<i>Active</i>	<i>True</i>

La tabla tendrá su correspondiente *TDataSource*:

	Propiedad	Valor
<i>dsPedidos</i>	<i>DataSet</i>	<i>tbPedidos</i>

Añada, finalmente, un par de rejillas de datos (*TDBGrid*, en la página *Data Access*), y modifique sus propiedades *DataSource* para que apunten a los dos componentes correspondientes. Ponga entonces un botón en algún sitio del formulario para efectuar la suma de los pedidos. Y pruebe este algoritmo inicial, en respuesta al evento *OnClick* del botón:

```
// PRIMERA VARIANTE: ;;MUY INEFICIENTE!!!
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    DWORD Tiempo = GetTickCount();
    Currency Total = 0;
    tbClientes->First();
    while (! tbClientes->Eof)
    {
        if (tbClientes->FieldValues["COUNTRY"] != "US")
        {
            tbPedidos->First();
            while (! tbPedidos->Eof)
            {
                Total += tbPedidos->FieldValues["ITEMSTOTAL"];
                tbPedidos->Next();
            }
        }
        tbClientes->Next();
    }
    ShowMessage(Format("%m: %d",
        ARRAYOFCONST((Total, GetTickCount() - Tiempo))));
}
```

He tenido que adelantarme un poco al orden de exposición del libro. Estoy acce-

diendo a los valores de los campos de las tablas mediante el método más sencillo ... y más ineficiente. Me refiero a estas instrucciones:

```
if (tbClientes->FieldValues["COUNTRY"] != "US")
// ...
    Total += tbPedidos->FieldValues["ITEMSTOTAL"];
```

Por el momento, debe saber que con esta técnica se obtiene el valor del campo como un tipo *Variant*, de modo que en dependencia del contexto en que se emplee el valor, C++ Builder realizará la conversión de tipos adecuada.

Habría observado el comentario que encabeza el primer listado de esta sección. Si ha seguido el ejemplo y pulsado el botón, comprenderá por qué es ineficiente ese código. ¡Cada vez que se mueve una fila, las rejillas siguen el movimiento! Y lo que más tiempo consume es el dibujo en pantalla. Sin embargo, usted ya conoce los métodos *EnableControls* y *DisableControls*, que desactivan las notificaciones a los controles visuales ¿Por qué no utilizarlos?

```
// SEGUNDA VARIANTE: ;;; INCORRECTA!!!
void __fastcall TForm1.Button1Click(TObject *Sender)
{
    DWORD Tiempo = GetTickCount();
    Currency Total = 0;
    tbClientes->DisableControls(); // ← NUEVO
    tbPedidos->DisableControls(); // ← NUEVO
    try
    {
        tbClientes->First();
        while (! tbClientes->Eof)
        {
            if (tbClientes->FieldValues["COUNTRY"] != "US")
            {
                tbPedidos->First();
                while (! tbPedidos->Eof)
                {
                    Total += tbPedidos->FieldValues["ITEMSTOTAL"];
                    tbPedidos->Next();
                }
            }
            tbClientes->Next();
        }
    }
    finally
    {
        tbPedidos->EnableControls(); // ← NUEVO
        tbClientes->EnableControls(); // ← NUEVO
    }
    ShowMessage(Format("%m: %d",
        ARRAYOFCONST((Total, GetTickCount() - Tiempo))));
}
```

Ahora el algoritmo sí va rápido, ¡pero devuelve un resultado a todas luces incorrecto! Cuando llamamos a *DisableControls* estamos desconectando el mecanismo de notifica-

ción de cambios de la tabla a sus controles visuales ... y también a las tablas que dependen en relaciones *master/detail*. Por lo tanto, se mueve la tabla de clientes, pero la tabla de pedidos no modifica el conjunto de filas activas cada vez que se selecciona un cliente diferente.

¿Quiere una solución que funcione en cualquier versión de C++ Builder? Es muy sencilla: utilice dos componentes *TDataSource* acoplados a la tabla de clientes. Traiga un nuevo componente de este tipo, *DataSource1*, y cambie su propiedad *DataSet* al valor *tbClientes*. Entonces, haga que la propiedad *DataSource* de *DBGrid1* apunte a *DataSource1* en vez de a *dsClientes*. Por último, modifique el algoritmo de iteración del siguiente modo:

```
// TERCERA VARIANTE: ;;;AL FIN BIEN!!!
void __fastcall TForm1.Button1Click(TObject *Sender)
{
    DWORD Tiempo = GetTickCount();
    Currency Total = 0;
    DataSource1->Enabled = False; // ← NUEVO
    tbPedidos->DisableControls();
    try
    {
        tbClientes->First();
        while (! tbClientes->Eof)
        {
            if (tbClientes->FieldValues["COUNTRY"] != "US")
            {
                tbPedidos->First();
                while (! tbPedidos->Eof)
                {
                    Total += tbPedidos->FieldValues["ITEMSTOTAL"];
                    tbPedidos->Next();
                }
            }
            tbClientes->Next();
        }
    }
    finally
    {
        tbPedidos->EnableControls();
        DataSource1->Enabled = True; // ← NUEVO
    }
    ShowMessage(Format("%m: %d",
        ARRAYOFCONST((Total, GetTickCount() - Tiempo))));
}
```

Ahora, el flujo de notificaciones se corta al nivel de una fuente de datos particular, no al nivel general de la tabla. De esta forma, *tbClientes* sigue enviando notificaciones a sus dos fuentes de datos asociadas. La fuente de datos *dsClientes* propaga estas notificaciones a los objetos que hacen referencia a ella: en este caso, la tabla dependiente *dsPedidos*. Pero *DataSource1* se inhabilita temporalmente para que la rejilla asociada no reciba notificaciones de cambio.

¿Quiere una solución que funciona solamente a partir de C++ Builder 4? En esta versión se introduce la propiedad *BlockReadSize*. Cuando el valor de la misma es mayor que cero, el conjunto de datos entra en un estado especial: la propiedad *State*, que veremos en la siguiente sección, toma el valor *dsBlockRead*. En este estado, las notificaciones de movimiento se envían solamente a las relaciones *master/detail*, pero no a los controles de datos. Parece ser que también se mejora la eficiencia de las lecturas, porque se leen simultáneamente varios registros por operación. Hay que tener en cuenta, sin embargo, dos inconvenientes:

- La única operación de navegación que funciona es *Next*.
- Al parecer, la modificación de esta propiedad en una tabla de detalles no funciona correctamente en la versión 4.0.

Teniendo en cuenta estas advertencias, nuestro algoritmo pudiera escribirse de esta forma alternativa en C++ Builder 4:

```
// CUARTA VARIANTE: ;;;SOLO VERSION 4!!!
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    DWORD Tiempo = GetTickCount();
    Currency Total = 0;
    // Se llama a First antes de modificar BlockReadSize
    tbClientes->First();
    tbClientes->BlockReadSize = 10; // ← NUEVO
    tbPedidos->DisableControls(); // Se mantiene
    try
    {
        while (! tbClientes->Eof)
        {
            if (tbClientes->FieldValues["COUNTRY"] != "US")
                begin
                    tbPedidos->First();
                    while (! tbPedidos->Eof)
                    {
                        Total += tbPedidos->FieldValues["ITEMSTOTAL"];
                        tbPedidos->Next();
                    }
                }
            tbClientes->Next();
        }
    }
    finally
    {
        tbPedidos->EnableControls();
        tbClientes->BlockReadSize = 0; // ← NUEVO
    }
    ShowMessage(Format("%m: %d",
        ARRAYOFCONST((Total, GetTickCount() - Tiempo))));
}
```

Ahora que ya sabe cómo realizar un doble recorrido sobre un par de tablas en relación *master/detail*, le aconsejo que solamente programe este tipo de algoritmos

cuando esté trabajando con bases de datos de escritorio. Si está utilizando una base de datos cliente/servidor, la ejecución de una consulta es incomparablemente más rápida. Puede comprobarlo.

El estado de un conjunto de datos

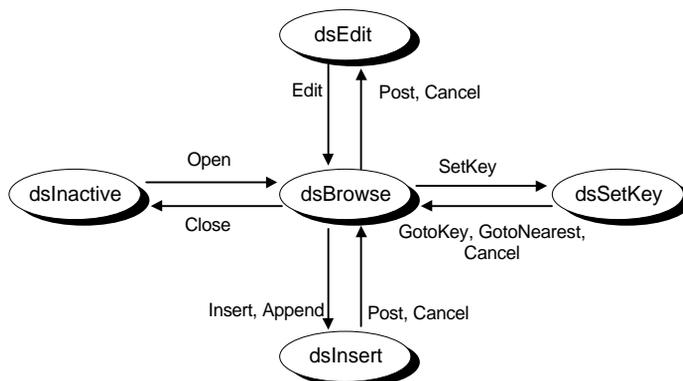
Una de las propiedades más importantes de los conjuntos de datos es *State*, cuya declaración es la siguiente:

```
enum TDataSetState = {dsInactive, dsBrowse, dsEdit, dsInsert,
    dsSetKey, dsCalcFields, dsFilter, dsNewValue, dsOldValue,
    dsCurValue, dsBlockRead, dsInternalCalc};

__property TDataSetState State;
```

El estado de un conjunto de datos determina qué operaciones se pueden realizar sobre el mismo. Por ejemplo, en el estado de exploración, *dsBrowse*, no se pueden realizar asignaciones a campos. Algunas operaciones cambian su semántica de acuerdo al estado en que se encuentre el conjunto de datos. El método *Post*, por ejemplo, graba una nueva fila si el estado es *dsInsert* en el momento de su aplicación, pero modifica la fila activa si el estado es *dsEdit*.

La propiedad *State* es una propiedad de sólo lectura, por lo que no podemos cambiar de estado simplemente asignando un valor a ésta. Incluso hay estados a los cuales el programador no puede llegar explícitamente. Tal es el caso del estado *dsCalcFields*, al cual se pasa automáticamente cuando existen campos calculados en la tabla; estos campos se estudiarán en el capítulo siguiente. Las transiciones de estado que puede realizar el programador se logran mediante llamadas a métodos. El siguiente diagrama, todo un clásico de los libros de C++ Builder, muestra los diferentes estados de un conjunto de datos a los que puede pasar explícitamente el programador, y las transiciones entre los mismos:



En el diagrama no se han representado los estados internos e inaccesibles. Los estados *dsUpdateNew* y *dsUpdateOld*, *dsOldValue*, *dsNewValue* y *dsCurrentValue* son estados utilizados internamente por la VCL, y el programador nunca encontrará que una rutina programada por él se está ejecutando con una tabla en uno de estos estados. En cambio, aunque el programador nunca coloca una tabla de forma explícita en los estados *dsCalcFields* y *dsFilter*, aprovecha estos estados durante la respuestas a un par de eventos, *OnCalcFields* y *OnFilterRecord*. El primero de estos eventos se utiliza para asignar valores a campos calculados; el segundo evento permite trabajar con un subconjunto de filas de una tabla, y lo estudiaremos en el capítulo sobre métodos de búsqueda.

La comprensión de los distintos estados de un conjunto de datos, los métodos y los eventos de transición son fundamentales para poder realizar actualizaciones en bases de datos. Más adelante volveremos necesariamente sobre este tema.

Acceso a campos

LOS COMPONENTES DE ACCESO A CAMPOS son parte fundamental de la estructura de la VCL. Estos objetos permiten manipular los valores de los campos, definir formatos de visualización y edición, y realizar ciertas validaciones básicas. Sin ellos, nuestros programas tendrían que trabajar directamente con la imagen física del *buffer* del registro activo en un conjunto de datos. Afortunadamente, C++ Builder crea campos aún cuando a nosotros se nos olvida hacerlo. En este capítulo estudiaremos las clases de campos y sus propiedades, concentrándonos en los tipos de campos “simples”, y en el uso del Diccionario de Datos para acelerar la configuración de campos en tiempo de diseño. En capítulos posteriores tendremos oportunidad de estudiar los campos BLOB y los correspondientes a las nuevas extensiones orientadas a objetos de Oracle 8.

Creación de componentes de campos

Por mucho que busquemos, nunca encontraremos los componentes de acceso a campos en la Paleta de Componentes. El quid está en que estos componentes se vinculan al conjunto de datos (tabla o consulta) al cual pertenecen, del mismo modo en que los ítems de menú se vinculan al objeto de tipo *TMainMenu* ó *TPopupMenu* que los contiene. Siguiendo la analogía con los menús, para crear componentes de campos necesitamos realizar una doble pulsación sobre una tabla para invocar al *Editor de Campos* de C++ Builder. Este Editor se encuentra también disponible en el menú local de las tablas como el comando *Fields editor*.

Antes de explicar el proceso de creación de campos necesitamos aclarar una situación: podemos colocar una tabla en un formulario, asociarle una fuente de datos y una rejilla, y echar a andar la aplicación resultante. ¿Para qué queremos campos entonces? Bueno, aún cuando no se han definido componentes de acceso a campos explícitamente para una tabla, estos objetos están ahí, pues han sido creados automáticamente por C++ Builder. Si durante la apertura de una tabla se detecta que el usuario no ha definido campos en tiempo de diseño, la VCL crea objetos de acceso de forma implícita. Por supuesto, estos objetos reciben valores por omisión para sus propiedades, que quizás no sean los que deseamos.

Precisamente por eso creamos componentes de campos en tiempo de diseño: para poder controlar las propiedades y eventos relacionados con los mismos. La creación en tiempo de diseño no nos hace malgastar memoria adicional en tiempo de ejecución, pues los componentes se van a crear de una forma u otra. Pero sí tenemos que contar con el aumento de tamaño del fichero *djfm*, que es donde se va a grabar la configuración persistente de los valores iniciales de las propiedades de los campos. Este es un factor a tener en cuenta, como veremos más adelante.



El Editor de Campos es una ventana de ejecución no modal; esto quiere decir que podemos tener a la vez en pantalla distintos conjuntos de campos, correspondiendo a distintas tablas, y que podemos pasar sin dificultad de un Editor a cualquier otra ventana, en particular, al Inspector de Objetos. Para realizar casi cualquier acción en el Editor de Campos hay que pulsar el botón derecho del ratón y seleccionar el comando de menú adecuado. Tenemos además una pequeña barra de navegación en la parte superior del Editor. Esta barra no está relacionada en absoluto con la edición de campos, sino que es un medio conveniente de mover, en tiempo de diseño, el cursor o fila activa de la tabla asociada.

Añadir componentes de campos es muy fácil, pues basta con ejecutar el comando *Add fields* del menú local. Se presenta entonces un cuadro de diálogo con una lista de los campos físicos existentes en la tabla y que todavía no tienen componentes asociados. Esta lista es de selección múltiple, y selecciona por omisión todos los campos. Es aconsejable crear componentes para todos los campos, aún cuando no tengamos en mente utilizar algunos campos por el momento. La explicación tiene que ver también con el proceso mediante el cual la VCL crea los campos. Si al abrir la tabla se detecta la presencia de al menos un componente de campo definido en tiempo de diseño, C++ Builder no intenta crear objetos de campo automáticamente. El resultado es que estos campos que dejamos sin crear durante el diseño *no existen* en lo que concierne a C++ Builder.

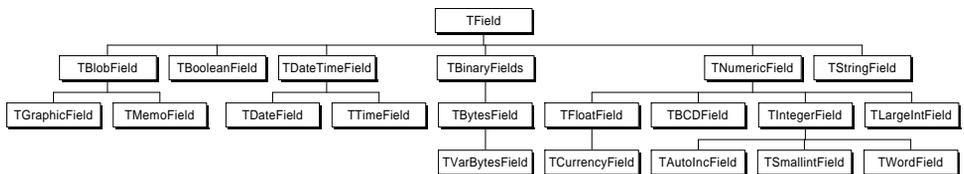
Add fields...	Ctrl+A
New field...	Ctrl+N
Add all fields	Ctrl+F
Cut	Ctrl+X
Copy	Ctrl+C
Paste	Ctrl+V
Delete	Del
Select all	Ctrl+L
Retrieve attributes	Ctrl+R
Save attributes	Ctrl+S
Save attributes as...	Ctrl+E
Associate attributes...	Ctrl+D
Unassociate attributes	Ctrl+U

Esta operación puede repetirse más adelante, si añadimos nuevos campos durante una reestructuración de la tabla, o si modificamos la definición de un campo. En este último caso, es necesario destruir primeramente el viejo componente antes de añadir el nuevo. Para destruir un componente de campo, sólo es necesario seleccionarlo en el Editor de Campos y pulsar la tecla SUPR.

El comando *Add all fields*, del menú local del Editor de Campos, es una novedad de C++ Builder 4 para acelerar la configuración de campos.

Clases de campos

Una vez creados los componentes de campo, podemos seleccionarlos en el Inspector de Objetos a través de la lista de objetos, o mediante el propio Editor de Campos. Lo primero que llama la atención es que, a diferencia de los menús donde todos los comandos pertenecen a la misma clase, *TMenuItem*, aquí cada componente de acceso a campo puede pertenecer a una clase distinta. En realidad, todos los componentes de acceso a campos pertenecen a una jerarquía de clases derivada por herencia de una clase común, la clase *TField*. El siguiente diagrama muestra esta jerarquía:

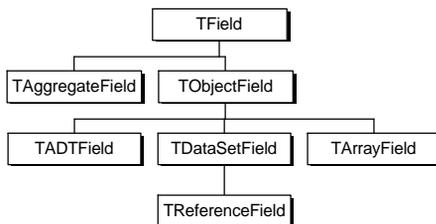


De todos estos tipos, *TField*, *TNumericField* y *TBinaryField* nunca se utilizan directamente para crear instancias de objetos de campos; su papel es servir de ancestro a campos de tipo especializado. La correspondencia entre los tipos de las variables de campos y los tipos de las columnas es la siguiente:

Tipo de campo	dBase	Paradox	InterBase
<i>TStringField</i>	char	alpha	char, varchar
<i>TIntegerField</i>		longint	int, long
<i>TAutoIncField</i>		autoinc	
<i>TWordField</i>			
<i>TSmallIntField</i>	number	shortint	short
<i>TBCDField</i>		bcd	
<i>TFloatField</i>	float, number	number	float, double
<i>TCurrencyField</i>		money	
<i>TBooleanField</i>	logical	logical	
<i>TDateField</i>	date	date	
<i>TTimeField</i>		time	
<i>TDateTimeField</i>		timestamp	date
<i>TBlobField</i>	ole, binary	fmtmemo, ole, binary	blob
<i>TGraphicField</i>		graphic	
<i>TMemoField</i>	memo	memo	text blob
<i>TBytesField</i>		bytes	
<i>TVarBytesField</i>			

Algunos tipos de campos se asocian con las clases de campos de C++ Builder en dependencia de su tamaño y precisión. Tal es el caso de los tipos **number** de dBase, y de **decimal** y **numeric** de InterBase.

A la jerarquía de clases que hemos mostrado antes, C++ Builder 4 añade un par de ramas:



La clase *TAggregateField* permite definir campos agregados con cálculo automático en conjuntos de datos clientes: sumas, medias, máximos, mínimos, etc. Tendremos que esperar un poco para examinarlos. En cuanto a la jerarquía que parte de la clase abstracta *TObjectField*, sirve para representar los nuevos tipos de datos orientados a objetos de Oracle 8: objetos incrustados (*TADTField*), referencias a objetos (*TReferenceField*), vectores (*TArrayField*) y campos de tablas anidadas (*TDataSetField*).

Aunque InterBase permite definir campos que contienen matrices de valores, las versiones actuales de la VCL y del BDE no permiten tratarlos como tales directamente.

Nombre del campo y etiqueta de visualización

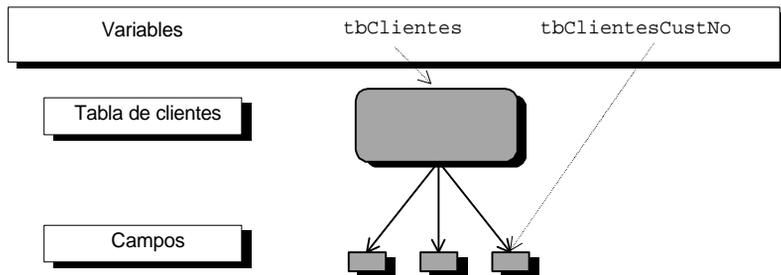
Existen tres propiedades de los campos que muchas veces son confundidas entre sí por el programador. Son las propiedades *Name*, *FieldName* y *DisplayLabel*. La primera es, como sucede con casi todos los componentes, el nombre de la variable de campo, o sea, del puntero al objeto. *FieldName* es el nombre de la columna de la tabla a la que se refiere el objeto de campo. Y *DisplayLabel* es un texto descriptivo del campo, que se utiliza, entre otras cosas, como encabezamiento de columna cuando el campo se muestra en una rejilla de datos.

De estas propiedades, *FieldName* es la que menos posibilidades nos deja: contiene el nombre de la columna, y punto. Por el contrario, *Name* se deduce inicialmente a partir del nombre de la tabla y del nombre de la columna. Si el nombre de tabla es *tbClientes* y el nombre del campo (*FieldName*) es *CustNo*, el nombre que C++ Builder le asigna a *Name*, y por consiguiente a la variable que apunta al campo, es *tbClientesCustNo*, la concatenación de ambos nombres.

Esto propicia un error bastante común entre los programadores, pues muchas veces escribimos por inercia, pensando en un esquema *tabla.campo*:

```
tbClientes->CustNo    // ;;;INCORRECTO!!!
```

El siguiente gráfico puede ayudar a comprender mejor la relación entre los nombres de variables y los componentes de tablas y de campos:



La asignación automática de nombres de componentes de campos nos plantea un problema práctico: el tamaño del fichero *dflm* crece desmesuradamente. Tomemos por ejemplo una aplicación pequeña que trabaje con diez tablas, y supongamos que cada tabla tiene diez campos; éstas son estimaciones a la baja. Entonces, tendremos cien componentes de campos, y cada componente tendrá un nombre kilométrico que estará ocupando espacio en el fichero *dflm* y luego en la memoria, en tiempo de ejecución. Es por eso que buscando un menor tiempo de carga de la aplicación, pues la memoria no es una consideración primordial en estos días, tengo la costumbre de renombrar los componentes de campos con el propósito de disminuir la longitud de los nombres en lo posible, sin caer en ambigüedades. Por ejemplo, el nombre de

componente *tbClientesCustNo* puede abreviarse a algo así como *tbClCustNo*; ya sé que son sólo seis letras menos, pero multiplíquelas por cien y verá.

Acceso a los campos por medio de la tabla

Aunque la forma más directa, segura y eficiente de acceder a un campo es crear el componente en tiempo de diseño y hacer uso de la variable asociada, es también posible llegar indirectamente al campo a través de la tabla a la cual pertenece. Estas son las funciones y propiedades necesarias:

```
TField* __fastcall TDataSet::FieldByName(const AnsiString Nombre);
__property TFields* TDataSet::Fields;
```

La clase *TFields*, por su parte, tiene las siguientes propiedades principales:

```
__property int TFields::Count;
__property TField* TFields::Fields[int Index];
```

La definición que he mostrado de *Fields* es la que corresponde a la versión 4 de la VCL. Antes, esta propiedad se definía del siguiente modo:

```
__property TField* TDataSet::Fields[int Index];
```

Los cambios se han introducido por culpa de la aparición de los tipos de objetos de Oracle 8, y del soporte para campos agregados de Midas 2.

Con *FieldByName* podemos obtener el componente de campo dado su nombre, mientras que con *Fields* lo obtenemos si conocemos su posición. Está claro que esta última propiedad debe utilizarse con cautela, pues si la tabla se reestructura cambian las posiciones de las columnas. Mediante *FieldByName* y *Fields* obtenemos un objeto de tipo *TField*, la clase base de la jerarquía de campos. Por lo tanto, no se pueden utilizar directamente las propiedades específicas de los tipos de campos más concretos sin realizar una conversión de tipo. A esto volveremos a referirnos. Mientras tanto, he aquí una muestra de cómo se accede a un campo dada su posición en C++ Builder 3 y 4:

C++ Builder 3	C++ Builder 4
Table1->Fields[0]->FieldName	Table1->Fields->Fields[0]->FieldName

Si a la función *FieldByName* le pasamos un nombre inexistente de campo, se produce una excepción, por lo cual no debemos utilizar esta función si lo que queremos es saber si el campo existe o no. Para esto último contamos con la función *FindField*, que devuelve el puntero al objeto si éste existe, o el puntero vacío si no:

```
TField* __fastcall TDataSet::FindField(const AnsiString Nombre);
```

Recuerde que el componente de campo puede haber sido creado explícitamente por usted en tiempo de diseño, pero que si no ha realizado esta acción, C++ Builder construye automáticamente estos objetos al abrir el conjunto de datos.

Extrayendo información de los campos

Un componente de campo contiene los datos correspondientes al valor almacenado en la columna asociada de la fila activa de la tabla, y la operación más frecuente con un campo es extraer o modificar este valor. La forma más segura y eficiente es, una vez creados los campos persistentes con la ayuda del Editor de Campos, utilizar las variables generadas y la propiedad *Value* de las mismas. Esta propiedad se define del tipo apropiado para cada clase concreta de campo. Si el campo es de tipo *TStringField*, su propiedad *Value* es de tipo *AnsiString*; si el campo es de tipo *TBooleanField*, el tipo de *Value* es **bool**.

```
ShowMessage(Format("%d-%s", ARRAYOFCONST(
    (tbClientesCodigo->Value, // Un valor entero
    tbClientesNombre->Value)))); // Una cadena de caracteres
```

Si la referencia al campo es del tipo genérico *TField*, como las que se obtienen con la propiedad *Fields* y la función *FieldByName*, es necesario utilizar propiedades con nombres como *AsString*, *AsInteger*, *AsFloat*, etc., que aclaran el tipo de datos que queremos recuperar.

```
ShowMessage(
    IntToStr(tbClientes->FieldByName("Codigo")->AsInteger)
    + "-" + tbClientes->FieldByName("Nombre")->AsString);
```

Las propiedades mencionadas intentan siempre hacer la conversión del valor almacenado realmente al tipo especificado; cuando no es posible, se produce una excepción. Por ejemplo, en el caso anterior hubiéramos podido utilizar también la propiedad *AsString* aplicada al campo entero *Codigo*.

Ahora bien, existe un camino alternativo para manipular los datos de un campo: la propiedad *FieldValues* de *TDataSet*. La declaración de esta propiedad es la siguiente:

```
__property Variant TDataSet::FieldValues[AnsiString FieldName];
```

Como la propiedad devuelve valores variantes, no es necesario preocuparse demasiado por el tipo del campo, pues la conversión transcurre automáticamente:

```
ShowMessage(tbClientes->FieldValues["Codigo"]
    + "-" + tbClientes->FieldValues["Nombre"]);
```

También puede utilizarse *FieldValues* con una lista de nombres de campos separados por puntos y comas. En este caso se devuelve una matriz variante formada por los valores de los campos individuales:

```
System::Variant V = tbClientes->FieldValues["Codigo;Nombre"];
ShowMessage(V.GetElement(0) + "-" + V.GetElement(1));
```

En Delphi, *FieldValues* es la propiedad vectorial por omisión de los conjuntos de datos, por lo cual pueden aplicarse los corchetes directamente a una variable de tabla, como si ésta fuera un vector. Esta facilidad de uso ha propiciado la proliferación de esta propiedad en los ejemplos de la VCL. No obstante, debemos utilizar esta propiedad lo menos posible, pues tiene dos posibles puntos de fallo: puede que nos equivoquemos al teclear el nombre del campo (no se detecta el error sino en ejecución), y puede que nos equivoquemos en el tipo de retorno esperado.

Intencionalmente, todos los ejemplos que he mostrado leen valores desde las componentes de campos, pero no modifican este valor. El problema es que las asignaciones a campos sólo pueden efectuarse estando la tabla en alguno de los estados especiales de edición; en caso contrario, provocaremos una excepción. En el capítulo 26, sobre actualizaciones, se tratan estos temas con mayor detalle; un poco más adelante veremos cómo se pueden asignar valores a campos calculados.

Es útil saber también cuándo es nulo o no el valor almacenado en un campo. Para esto se utiliza la función *IsNull*, que retorna un valor de tipo **bool**.

Por último, si el campo es de tipo memo, gráfico o BLOB, no existe una propiedad simple que nos proporcione acceso al contenido del mismo. Más adelante explicaremos cómo extraer información de los campos de estas clases.

Las máscaras de formato y edición

El formato en el cual se visualiza el contenido de un campo puede cambiarse, para ciertos tipos de campos, por medio de una propiedad llamada *DisplayFormat*. Esta propiedad es aplicable a campos de tipo numérico, flotante y de fecha y hora; los campos de cadenas de caracteres no tienen una propiedad tal, aunque veremos en la próxima sección una forma de superar este “inconveniente”.

Si el campo es numérico o flotante, los caracteres de formato son los mismos que los utilizados por la función predefinida *FormatFloat*:

Carácter Significado

0	Dígito obligatorio
#	Dígitos opcionales
.	Separador decimal
,	Separador de millares
;	Separador de secciones

La peculiaridad principal de estas cadenas de formato es que pueden estar divididas hasta en tres secciones: una para los valores positivos, la siguiente para los negativos y la tercera sección para el cero. Por ejemplo, si *DisplayFormat* contiene la cadena "\$#,;(#.00);Cero", la siguiente tabla muestra la forma en que se visualizará el contenido del campo:

Valor del campo Cadena visualizada

12345	\$12.345
-12345	(12345.00)
0	Cero

Observe que la coma, el separador de millares americano, se traduce en el separador de millares nacional, y que lo mismo sucede con el punto. Otra propiedad relacionada con el formato, y que puede utilizarse cuando no se ha configurado *DisplayFormat*, es *Precision*, que establece el número de decimales que se visualizan por omisión. Tenga bien en cuenta que esta propiedad no limita el número de decimales que podemos teclear para el campo, ni afecta al valor almacenado finalmente en el mismo.

Cuando el campo es de tipo fecha, hora o fecha y hora, el significado de las cadenas de *DisplayFormat* coincide con el del parámetro de formato de la función *FormatDateTime*. He aquí unos pocos aunque no exhaustivos ejemplos de posibles valores de esta propiedad:

Valor de *DisplayFormat* Ejemplo de resultado

<i>dd-mm-yy</i>	04-07-64
<i>dddd, d "de" mmmm "de" yyyy</i>	sábado, 26 de enero de 1974
<i>hh:mm</i>	14:05
<i>h:mm am/pm</i>	2:05 pm

La preposición “de” se ha tenido que encerrar entre dobles comillas, pues en caso contrario la rutina de conversión interpreta la primera letra como una indicación para poner el día de la fecha.

Si el campo es de tipo lógico, de clase *TBooleanField*, la propiedad *DisplayValues* controla su formato de visualización. Esta propiedad, de tipo *AnsiString*, debe contener un par de palabras o frases separadas por un punto y coma; la primera frase corresponde al valor verdadero y la segunda al valor falso:

```
tbDiarioBuenTiempo->DisplayValues =
    "Un tiempo maravilloso;Un día horrible";
```

Por último, la edición de los campos de tipo cadena, fecha y hora puede controlarse mediante la propiedad *EditMask*. Los campos de tipo numérico y flotante no permiten esta posibilidad. Además, la propiedad *EditFormat* que introducen estos campos no sirve para este propósito, pues indica el formato inicial que se le da al valor numérico cuando comienza su edición. Por ejemplo, usted desea que el campo *Precio* se muestre en dólares, y utiliza la siguiente máscara de visualización en *DisplayFormat*:

```
$#0,.00
```

Entonces, para eliminar los caracteres superfluos de la edición, como el signo de dólar y los separadores de millares, debe asignar *#0.00* a la propiedad *EditFormat* del campo.

Los eventos de formato de campos

Cuando no bastan las máscaras de formato y edición, podemos echar mano de dos eventos pertenecientes a la clase *TField*: *OnGetText* y *OnSetText*. El evento *OnGetText*, por ejemplo, es llamado cada vez que C++ Builder necesita una representación visual del contenido de un campo. Esto sucede en dos circunstancias diferentes: cuando se está visualizando el campo de forma normal, y cuando hace falta un valor inicial para la edición del campo. El prototipo del evento *OnGetText* es el siguiente:

```
typedef void __fastcall (__closure *TFieldGetTextEvent)
    (TField *Sender, AnsiString &Text, bool DisplayText);
```

Un manejador de eventos para este evento debe asignar una cadena de caracteres en el parámetro *Text*, teniendo en cuenta si ésta se necesita para su visualización normal (*DisplayText* igual a *True*) o como valor inicial del editor (en caso contrario).

Inspirado en la película de romanos que pasaron ayer por la tele, he desarrollado un pequeño ejemplo que muestra el código del cliente en números romanos:

```
void __fastcall TForm1::tbClientesCustNoGetText(TField *Sender,
    AnsiString &Text, bool DisplayText)
{
    static AnsiString Unidades[10] =
        {"", "I", "II", "III", "IV", "V", "VI", "VII", "VIII", "IX"};
    static AnsiString Decenas[10] =
        {"", "X", "XX", "XXX", "XL", "L", "LX", "LXX", "LXXX", "XC"};
    static AnsiString Centenas[10] =
        {"", "C", "CC", "CCC", "CD", "D", "DC", "DCC", "DCCC", "CM"};
    static AnsiString Miles[4] =
        {"", "M", "MM", "MMM"};
```

```

if (Sender->AsInteger > 3999)
    Text = "Infinitum";
    // Hay que ser consecuentes con el lenguaje
else
{
    int i = Sender->AsInteger;
    Text = Miles[i / 1000] + Centenas[i / 100 % 10] +
        Decenas[i / 10 % 10] + Unidades[i % 10];
}
}

```

Si a algún usuario se le ocurriera la peregrina idea de teclear sus datos numéricos como números romanos, el evento adecuado para programar esto sería *OnSetText*. El prototipo del evento es el siguiente:

```

typedef void __fastcall (__closure *TFieldSetTextEvent)
(TField *Sender, const AnsiString Text);

```

Este evento es utilizado con frecuencia para realizar cambios sobre el texto tecleado por el usuario para un campo, antes de ser asignado al mismo. Por ejemplo, un campo de tipo cadena puede convertir la primera letra de cada palabra a mayúsculas, como sucede en el caso de los nombres propios. Un campo de tipo numérico puede eliminar los separadores de millares que un usuario puede colocar para ayudarse en la edición. Como este evento se define para el campo, es independiente de la forma en que se visualice dicho campo y, como veremos al estudiar los módulos de datos, formará parte de las reglas de empresa de nuestro diseño.

Para ilustrar el uso del evento *OnSetText*, aquí está el manejador que lleva a mayúsculas la primera letra de cada palabra de un nombre:

```

void __fastcall TForm1::tbClientesCompanySetText(TField *Sender,
const AnsiString Text)
{
    AnsiString S = Text;
    for (int i = 1; i <= S.Length(); i++)
        if (i == 1 || S[i-1] == ' ')
            CharUpperBuff(&S[i], 1);
    Sender->AsString = S;
}

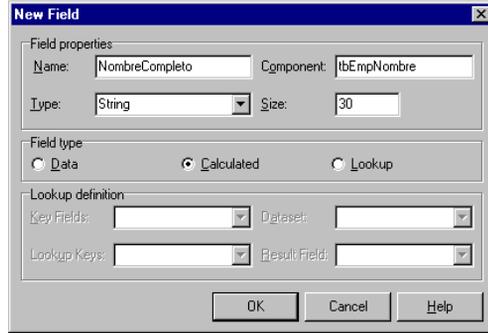
```

Otra situación práctica en la que *OnSetText* puede ayudar es cuando se necesite completar una entrada incompleta, en el caso de que el conjunto de valores a teclear esté limitado a ciertas cadenas.

Campos calculados

Una potente característica de C++ Builder es la que permite crear *campos calculados*, que no corresponden a campos definidos “físicamente” sobre la tabla o consulta

base, sino que se calculan a partir de los campos “reales”. Por ejemplo, la antigüedad de un trabajador puede deducirse a partir de su fecha de contrato y de la fecha actual. No tiene sentido, en cambio, almacenar físicamente este dato, pues tendría que actualizarse día a día. Otro candidato a campo calculado es el nombre completo de una persona, que puede deducirse a partir del nombre y los apellidos; es conveniente, en muchos casos, almacenar nombre y apellidos en campos independientes para permitir operaciones de búsqueda sobre los mismos por separado.



Para definir campos calculados, utilizamos el comando *New field*, del menú local del Editor de Campos. El cuadro de diálogo que aparece nos sirve para suministrar el nombre que le vamos a dar al campo, el nombre de la variable asociada al objeto, el tipo de campo y su longitud, si el campo contendrá cadenas de caracteres. También hay que aclarar que el campo es *Calculated*.

Observe que en ninguna parte de este cuadro de diálogo se ha escrito algo parecido a una fórmula. El algoritmo de cálculo del campo se especifica realmente durante la respuesta al evento *OnCalcFields* de la tabla a la cual pertenece el campo. Durante el intervalo de tiempo que dura la activación de este evento, la tabla se sitúa automáticamente en el estado *dsCalcFields*. En este estado, no se permiten las asignaciones a campos que no sean calculados: no se puede “aprovechar” el evento para realizar actualizaciones sobre campos físicos de la tabla, pues se produce una excepción. Tampoco debemos mover la fila activa, pues podemos provocar una serie infinita de llamadas recursivas; esto último, sin embargo, no lo controla C++ Builder. El evento *OnCalcFields* se lanza precisamente cuando se cambia la fila activa de la tabla. También puede lanzarse cuando se realiza alguna modificación en los campos de una fila, si la propiedad lógica *AutoCalcFields* del conjunto de datos vale *True*.

Para la tabla *employee.db* que podemos encontrar en el alias *bcdemos* de C++ Builder, podemos definir un par de campos calculados con esta estructura:

Campo	Tipo	Tamaño
<i>NombreCompleto</i>	<i>String</i>	30
<i>Antigüedad</i>	<i>Integer</i>	

Esto se realiza en el Editor de Campos. Después seleccionamos la tabla de empleados, y mediante el Inspector de Objetos creamos un manejador para su evento *OnCalcFields*; suponemos que el nombre del componente de tabla es *tbEmpleados*:

```
void __fastcall TmodDatos::tbEmpleadosCalcFields(TDataSet *Sender)
{
    tbEmpleadosNombreCompleto->Value =
        tbEmpleadosLastName->Value + ", " +
        tbEmpleadosFirstName->Value;
    if (! tbEmpleadosHireDate->IsNull)
        tbEmpleadosAntiguedad->Value =
            int(Date() - tbEmpleadosHireDate->Value) / 365;
}
```

Hay que tener cuidado con los campos que contienen valores nulos. En principio basta con que uno de los campos que forman parte de la expresión sea nulo para que la expresión completa también lo sea, en el caso de una expresión que no utilice operadores lógicos binarios (**and/or**). Esto es lo que hacemos antes de calcular el valor de la antigüedad, utilizando la propiedad *IsNull*. Por el contrario, he asumido que el nombre y los apellidos no pueden ser nulos, por lo cual no me he tomado la molestia de comprobar este detalle.

Campos de búsqueda

Es muy frecuente encontrar tablas conectadas entre sí mediante una relación uno/muchos; ya hemos visto que esta dependencia se puede expresar en nuestros programas mediante una relación *master/detail* entre tablas. A veces es conveniente, sin embargo, considerar la inversa de esta relación. Por ejemplo, un cliente “posee” un conjunto de pedidos ¿Debemos por esto trabajar los pedidos como detalles de un cliente? Podemos hacerlo de esta manera, pero lo usual es que las altas de pedidos se realicen sobre una tabla sin restricciones. Desde el punto de vista de la tabla de pedidos, su relación con la tabla de clientes es una *relación de referencia*.

Teniendo el código de cliente que se almacena en la tabla de pedidos, es deseable poder obtener el nombre de la empresa representada por ese código. Se pueden crear campos calculados que realicen manualmente la búsqueda por medio de índices en la tabla a la cual se hace referencia. En los capítulos sobre índices y técnicas de búsqueda veremos cómo implementar esta búsqueda. Sin embargo, pocas veces es necesario llegar a estos extremos, pues desde la versión 2 de la VCL disponemos de los denominados *campos de búsqueda* (o, en inglés, *lookup fields*), para los cuales el sistema ejecuta automáticamente el algoritmo de traducción de referencias.

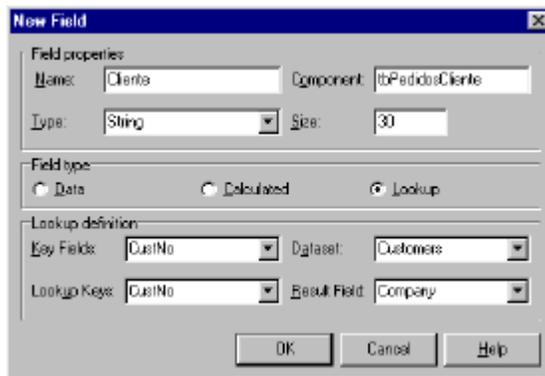
Los campos de búsqueda se crean por medio del comando de menú *New field* del menú local del Editor de Campos; se trata del mismo cuadro de diálogo que crea campos calculados. Pongamos por caso que queremos crear un campo que nos dé el

nombre del cliente asociado a un pedido. Utilizaremos ahora las tablas de pedidos (*tbPedidos*, asociada a *orders.db*) y de clientes (*tbClientes*, asociada a *customer.db*). Nos vamos a la tabla de pedidos, activamos el Editor de Campos y el comando *New field*. La información de la parte superior del cuadro de diálogo es la misma que para un campo calculado:

Campo	Tipo	Tamaño
<i>Cliente</i>	<i>String</i>	30

Después, tenemos que indicar el tipo de campo como *Lookup*; de este modo, se activan los controles de la parte inferior del diálogo. Estos son los valores que ha que suministrar, y su significado:

- Key fields* El campo o conjunto de campos que sirven de base a la referencia. Están definidos en la tabla base, y por lo general, aunque no necesariamente, tienen definida una clave externa. En este ejemplo, teclee *CustNo*.
- Dataset* Conjunto de datos en el cual se busca la referencia: use *tbClientes*.
- Lookup keys* Los campos de la tabla de referencia sobre los cuales se realiza la búsqueda. Para nuestro ejemplo, utilice *CustNo*; aunque es el mismo nombre que hemos tecleado en *Key fields*, esta vez nos estamos refiriendo a la tabla de clientes, en vez de la de pedidos.
- Result field* El campo de la tabla de referencia que se visualiza. Seleccione *Company*, el nombre de la empresa del cliente.



Un error frecuente es dejar a medias el diálogo de definición de un campo de búsqueda. Esto sucede cuando el programador inadvertidamente pulsa la tecla *INTRO*, pensando que de esta forma selecciona el próximo control de dicha ventana. Cuando esto sucede, no hay forma de volver al cuadro de diálogo para terminar la definición. Una posibilidad es eliminar la definición parcial y comenzar desde cero. La segunda consiste en editar directamente las propiedades del campo recién creado. La siguiente tabla enumera estas propiedades y su correspondencia con los controles del diálogo de definición:

Propiedad	Correspondencia
<i>Lookup</i>	Siempre igual a <i>True</i> para estos campos
<i>KeyFields</i>	Campos de la tabla en los que se basa la búsqueda (<i>Key fields</i>)
<i>LookupDataset</i>	Tabla de búsqueda (<i>Dataset</i>).
<i>LookupKeyFields</i>	Campos de la tabla de búsqueda que deben corresponder al valor de los <i>KeyFields</i> (<i>Lookup keys</i>)
<i>LookupResultField</i>	Campo de la tabla de búsqueda cuyo valor se toma (<i>Result field</i>)

La caché de búsqueda

A partir de la versión 3 de la VCL se han añadido propiedades y métodos para hacer más eficiente el uso de campos de búsqueda. Consideremos, por un momento, un campo que debe almacenar formas de pago. Casi siempre estas formas de pago están limitadas a un conjunto de valores determinados. Si utilizamos un conjunto especificado directamente dentro del código de la aplicación se afecta la extensibilidad de la misma, pues para añadir un nuevo valor hay que recompilar la aplicación. Muchos programadores optan por colocar los valores en una tabla y utilizar un campo de búsqueda para representar las formas de pago; en la columna correspondiente se almacena ahora el código asociado a la forma de pago. Desgraciadamente, este estilo de programación era bastante ineficiente en la VCL 2, sobre todo en entornos cliente/servidor.

Se puede y debe utilizar memoria caché para los campos de búsqueda si se dan las siguientes condiciones:

- La tabla de referencia contiene relativamente pocos valores.
- Es poco probable que cambie la tabla de referencia.

Para activar la caché de un campo de búsqueda se utiliza la propiedad *LookupCache*, de tipo lógico: asignándole *True*, los valores de referencia se almacenan en la propiedad *LookupList*, de la cual C++ Builder los extrae una vez inicializada automáticamente. Si ocurre algún cambio en la tabla de referencia mientras se ejecuta la aplicación, basta con llamar al método *RefreshLookupList*, sobre el componente de campo, para releer los valores de la memoria caché.

¿Son peligrosos los campos de búsqueda en la programación cliente/servidor? No, en general. Una alternativa a ellos es el uso de consultas basadas en encuentros (*joins*), que tienen el inconveniente de no ser actualizables intrínsecamente y el de ser verdaderamente peligrosas para la navegación, si el conjunto resultado es grande. El problema con los campos de búsqueda surge cuando se visualizan en un control *TDBLookupComboBox*, que permite la búsqueda incremental insensible a mayúsculas y minúsculas. La búsqueda se realiza con el método *Locate*, que

desempeña atrozmente este cometido en particular. Para más información, lea el capítulo 25, sobre la comunicación cliente/servidor.

Cuando estudiemos Midas, veremos otra alternativa para implementar eficientemente campos de búsqueda sobre tablas casi estáticas de pequeño tamaño, que consiste en mover al cliente el contenido de las tablas de referencia mediante el componente *TClientDataSet*.

El orden de evaluación de los campos

Cuando se definen campos calculados y campos de búsqueda sobre una misma tabla, los campos de búsqueda se evalúan antes que los campos calculados. Así, durante el algoritmo de evaluación de los campos calculados se pueden utilizar los valores de los campos de búsqueda.

Tomemos como ejemplo la tabla *items.db*, para la cual utilizaremos un componente de tabla *tbDetalles*. Esta tabla contiene líneas de detalles de pedidos, y en cada registro hay una referencia al código del artículo vendido (*PartNo*), además de la cantidad (*Qty*) y el descuento aplicado (*Discount*). A partir del campo *PartNo*, y utilizando la tabla *parts.db* como tabla de referencia para los artículos, puede crearse un campo de búsqueda *Precio* que devuelva el precio de venta del artículo en cuestión, extrayéndolo de la columna *ListPrice* de *parts.db*. En este escenario, podemos crear un campo calculado, de nombre *SubTotal* y de tipo *Currency*, que calcule el importe total de esa línea del pedido:

```
void __fastcall TmodDatos::tbDetallesCalcFields(TDataSet *DataSet)
{
    tbDetallesSubTotal->Value =
        tbDetallesQty->Value *                // Campo base
        tbDetallesPrecio->Value *            // Campo de búsqueda
        (1 - tbDetallesDiscount->Value / 100); // Campo base
}
```

Lo importante es que, cuando se ejecuta este método, ya C++ Builder ha evaluado los campos de búsqueda, y tenemos un valor utilizable en el campo *Precio*.

Otra información que es importante conocer acerca del orden de evaluación, es que la VCL evalúa los campos calculados antes de haber fijado el rango de filas de las tablas de detalles asociadas. Si la tabla *tbDetalles* está asociada a una tabla con cabeceras de pedidos, *tbPedidos*, por una relación *master/detail*, no es posible definir un campo calculado en la tabla de pedidos que sume los subtotales las líneas asociadas, extrayendo los valores de *tbDetalles*. Es necesario utilizar una tercera tabla auxiliar, que se refiera a la tabla de líneas de detalles, para buscar en esta última las líneas necesarias y realizar la suma.

Extensiones para los tipos de objetos de Oracle 8

Para poder manejar los nuevos tipos de datos de Oracle 8, C++ Builder 4 ha añadido cuatro nuevos tipos de campos, un nuevo tipo de conjunto de datos y ha efectuado modificaciones en el tipo *TDBGrid* para poder visualizar datos de objetos. Los cambios en el BDE son los siguientes:

- El parámetro *DLL32* del controlador de Oracle debe ser *sqlora8.dll*.
- *VENDOR INIT* debe ser *oci.dll*.
- *OBJECT MODE* debe valer *TRUE*.

Los nuevos tipos de campos son:

Tipo	Significado
<i>TADTField</i>	Para los objetos anidados
<i>TArrayField</i>	Representa un vector
<i>TDataSetField</i>	Tablas anidadas
<i>TReferenceField</i>	Contiene una referencia a un objeto compartido

Y el nuevo conjunto de datos es *TNestTable*, que representa al conjunto de filas contenidas en un campo de tipo *TDataSetField*.

Comencemos por el tipo de campo más sencillo: el tipo *TADTField*. Pongamos por caso que en la base de datos hemos definido la siguiente tabla:

```

create type TFraccion as object (
    Numerador    number(9) ,
    Denominador  number(9)
);
/

create table Probabilidades (
    Suceso       varchar2(30) not null primary key,
    Probabilidad TFraccion not null
);

```

En C++ Builder, traemos un *TTable*, lo conectamos a esta tabla y traemos todos los campos, mediante el comando *Add all fields*. Estos son los punteros a campos que son creados:

```

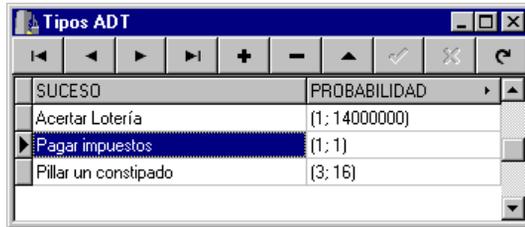
TTable *tbProb;
TStringField *tbProbSUCESO;
TADTField *tbProbPROBABILIDAD;
TFloatField *tbProbNUMERADOR;
TFloatField *tbProbDENOMINADOR;

```

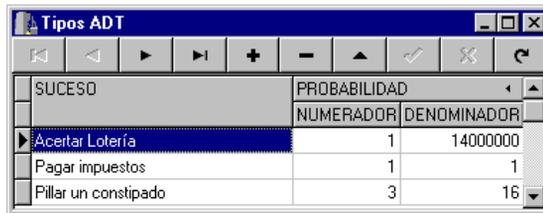
Es decir, podemos trabajar directamente con los campos básicos de tipos simples, o acceder a la columna que contiene el objeto mediante el tipo *TADTField*. Por ejemplo, todas estas asignaciones son equivalentes:

```
tbProbNUMERADOR->Value = 1;
tbProbPROBABILIDAD->FieldValues[0] = 1;
tbProbPROBABILIDAD->Fields->Fields[0]->AsInteger = 1;
tbProb->FieldValues["PROBABILIDAD.NUMERADOR"] = 1;
```

O sea, que hay más de un camino para llegar a Roma. Ahora mostraremos la tabla en una rejilla. Por omisión, este será el aspecto de la rejilla:



El campo ADT aparece en una sola columna. En sus celdas, el valor de campo se representa encerrando entre paréntesis los valores de los campos más simples; estas celdas no son editables. Las columnas de rejillas de C++ Builder 4, que estudiaremos en el momento apropiado, tienen una nueva propiedad *Expanded*, que en este caso vale *False*. Cuando se pulsa sobre la flecha que aparece a la derecha de la columna *Probabilidad*, la columna se expande en sus componentes, que sí se pueden modificar:



Un campo de tipo *TADTField* puede mostrarse, en modo sólo lectura, en cualquier otro control de datos, como un *TDBEdit*. Sin embargo, es más lógico que estos controles se conecten a los componentes simples del ADT. Por ejemplo, si quisiéramos editar la tabla anterior registro a registro, necesitaríamos tres controles de tipo *TDBEdit*, uno para el suceso, otro para el numerador y el último para el denominador.

Vamos ahora con las tablas anidadas y el campo *TDataSetField*. Este campo tampoco tiene previsto editarse directamente en controles de datos. Ahora bien, cuando una columna de una rejilla está asociada a uno de estos campos, al intentar una modifica-

ción aparece una ventana emergente con otra rejilla, esta vez asociada a las filas anidadas de la fila maestra activa, como en la siguiente imagen:



De dónde ha salido la tabla anidada? C++ Builder ha configurado un objeto de tipo *TNestTable* y lo ha acoplado a la propiedad *NestedDataSet* del campo *TDataSetField*. Sin embargo, es más frecuente que configuremos nosotros mismos un componente *TNestTable* para mostrar la relación uno/muchos explícitamente. La propiedad principal de este componente es *DataSetField*, y no tenemos que indicar base de datos ni nombre de tabla, pues estos parámetros se deducen a partir del campo.



Es conveniente tener un poco de precaución al editar tablas anidadas, pues no siempre se refresca correctamente su contenido al realizarse inserciones. ¿Motivo?, el hecho de que el BDE siga dependiendo de las claves primarias para identificar registros (vea el último capítulo de esta parte), cuando en este caso debería utilizar el identificador de fila, o *rowid*. Mi consejo es, mientras no aparezca otra técnica mejor, llamar al método *Post* de la tabla principal en el cuerpo de los eventos *BeforeInsert* y *AfterPost* de la tabla anidada.

Los campos de referencia, *TReferenceField*, son muy similares a las tablas anidadas; de hecho, esta clase descende por herencia de *TDataSetField*. La única diferencia consiste en que la tabla anidada que se le asocia contiene como máximo una sola fila. C++ Builder permite extraer de forma sencilla los valores del objeto asociado a la referencia, el equivalente del operador **deref** de Oracle:

```
ShowMessage("Cliente: " +
    tbPedidoRefCliente->Fields->Fields[0].AsString);
```

Sin embargo, para hacer que un campo de referencia apunte a un objeto existente necesitamos apoyarnos en una consulta o procedimiento almacenado que devuelva la referencia a dicho objeto. Antes habíamos mostrado un pequeño ejemplo en el que los pedidos tenían una referencia a una tabla de objetos clientes. Supongamos que en nuestra aplicación, al introducir un pedido, el usuario selecciona un cliente mediante una ventana emergente o algún mecanismo similar. Para asignar la referencia puede utilizarse el siguiente código:

```
std::auto_ptr<TQuery> query(new TQuery(NULL));
query->DatabaseName = tbPedidos->DatabaseName;
query->SQL->Add("select ref(Cli) from Clientes Cli ");
query->SQL->Add("where Cli.Nombre = " + tbClientesNombre->Value);
query->Open();
// Esta es la asignación deseada
tbPedidoCliente->Assign(query->Fields->Fields[0]);
```

Información sobre campos

C++ Builder hace distinción entre la información sobre los campos físicos de una tabla y los componentes de acceso a los mismos. La información sobre campos se encuentra en la propiedad *FieldDefs*, y puede utilizarse, además, para la creación dinámica de tablas. Esta propiedad pertenece a la clase *TFieldDefs*, que es básicamente una lista de objetos de tipo *TFieldDef*. Las propiedades principales de esta clase son *Count*, que es la cantidad de campos, e *Items*, que es una propiedad vectorial para obtener cada definición individual. En *TFieldDefs* tenemos el método *Update*, que lee la lista de definiciones de campos del conjunto de datos, incluso si éste se encuentra cerrado:

```
std::auto_ptr<TTable> table(new TTable(NULL));
table->DatabaseName = "BCDEMOS";
table->TableName = "employee.db";
// Leer las definiciones de campos
table->FieldDefs->Update();
// ...
```

Una vez que tenemos la lista de definiciones en memoria, podemos acceder a cada definición por medio de la propiedad *Items*. Estas son las propiedades principales de la clase *TFieldDef*, a la cual pertenecen los elementos individuales de la colección:

Propiedad	Significado
<i>Name</i>	El nombre del campo
<i>DataType</i>	El tipo del campo
<i>Size</i>	Tamaño del campo, si es aplicable
<i>Required</i>	¿Admite valores nulos?
<i>FieldClass</i>	Referencia a la clase correspondiente derivada de <i>TField</i>

El siguiente método coloca en una lista de cadenas, que puede ser la de un *TListBox* o un *TMemo*, la definición de los campos de una tabla, en formato parecido al de SQL:

```

char* DataTypeStr[] = {"Unknown", "VarChar", "Smallint", "Integer",
    "Word", "Boolean", "Float", "Currency", "BCD", "Date", "Time",
    "DateTime", "Bytes", "VarBytes", "AutoInc", "Blob", "Memo",
    "Graphic", "FmtMemo", "ParadoxOle", "DBaseOle", "TypedBinary",
    "Cursor", "FixedChar", "WideString", "Largeint", "ADT", "Array",
    "Reference", "DataSet"};

void __fastcall LeerDefinicion(const AnsiString ADB,
    const AnsiString ATable, TStrings *Lista)
{
    std::auto_ptr<TTable> table(new TTable(NULL));
    table->DatabaseName = ADB;
    table->TableName = ATable;
    table->FieldDefs->Update();
    Lista->Clear();
    Lista->Add("create table " + ATable + "(");
    for (int i = 0; i < table->FieldDefs->Count; i++)
    {
        TFieldDef *fd = table->FieldDefs->Items[i];
        AnsiString S = "    " + fd->Name + " " +
            DataTypeStr[fd->DataType];
        if (fd->Size != 0)
            S = S + "(" + IntToStr(fd->Size) + ")";
        if (fd->Required)
            AppendStr(S, " not null");
        if (i < table->FieldDefs->Count - 1)
            AppendStr(S, ",");
        Lista->Add(S);
    }
    Lista->Add(")");
}

```

Este procedimiento puede llamarse posteriormente del siguiente modo:

```

void __fastcall TForm1::MostrarInfoClick(TObject *Sender)
{
    LeerDefinicion("BCDEMOS", "EMPLOYEE", ListBox1->Items);
}

```

Creación de tablas

Aunque personalmente prefiero crear tablas mediante instrucciones SQL, es posible utilizar propiedades y métodos del componente *TTable* para esta operación. La razón de mi preferencia es que la VCL no ofrece mecanismos para la creación directa de restricciones de rango, de integridad referencial y valores por omisión para columnas; para esto, tenemos que utilizar llamadas directas al BDE, si tenemos que tratar con tablas Paradox, o utilizar SQL en las bases de datos que lo permiten.

De cualquier forma, las tablas simples se crean fácilmente, y la clave la tienen las propiedades *FieldDefs* e *IndexDefs*; esta última propiedad se estudiará con más profundidad en el capítulo sobre índices. La idea es llenar la propiedad *FieldDefs* mediante llamadas al método *Add*, y llamar al final al método *CreateTable*. Por ejemplo:

```
void __fastcall TForm1::CrearTabla()
{
    std::auto_ptr<TTable> table(new TTable(NULL));
    table->DatabaseName = "BCDEMOS";
    table->TableName = "Whiskeys";
    table->TableType = ttParadox;
    table->FieldDefs->Add("Código", ftAutoInc, 0, False);
    table->FieldDefs->Add("Marca", ftString, 20, True);
    table->FieldDefs->Add("Precio", ftCurrency, 0, True);
    table->FieldDefs->Add("Puntuación", ftSmallInt, 0, False);
    table->CreateTable();
}
```

El alias que asociamos a *DatabaseName* determina el formato de la tabla. Si el alias es estándar, tenemos que utilizar la propiedad *TableType* para diferenciar las tablas Paradox de las tablas dBase.

Sin embargo, nos falta saber cómo crear índices y claves primarias utilizando esta técnica. Necesitamos manejar la propiedad *IndexDefs*, que es en muchos sentidos similar a *FieldDefs*. En el capítulo sobre índices explicaremos el uso de dicha propiedad. Por ahora, adelantaremos la instrucción necesaria para crear un índice primario para la tabla anterior, que debe añadirse antes de la ejecución de *CreateTable*:

```
// ...
table->IndexDefs->Add("", "Código", TIndexOptions()<<ixPrimary);
// ...
```

Hasta C++ Builder 3 las definiciones de campos no eran visibles en tiempo de diseño, pero la versión 4 ha movido la declaración de la propiedad *FieldDefs* a la sección **published**. De esta forma, cuando queremos crear una tabla en tiempo de ejecución podemos ahorrarnos las llamadas al método *Add* de *FieldDefs*, pues esta propiedad ya viene configurada desde el tiempo de diseño. Hay dos formas de rellenar *FieldDefs*: entrar a saco en el editor de la propiedad, o leer las definiciones desde una tabla existente. Para esto último, ejecute el comando *Update table definition*, desde el menú de contexto de la tabla, que adicionalmente asigna *True* a su propiedad *StoreDefs*.

C++ Builder 4 añade una propiedad *ChildDefs* a los objetos de clase *TFieldDef*, para tener en cuenta a los campos ADT de Oracle 8.

Y como todo lo que empieza tiene que acabar, es bueno saber cómo eliminar y renombrar tablas. El método *DeleteTable* permite borrar una tabla. La tabla debe estar cerrada, y deberán estar asignadas las propiedades que especifican el nombre del

alias, el de la tabla, y el tipo de tabla, de ser necesario. Otro método relacionado es *RenameTable*.

```
void __fastcall TTable::RenameTable(const AnsiString NuevoNombre);
```

Este método solamente puede utilizarse con tablas Paradox y dBase, y permite renombrar en una sola operación todos los ficheros asociados a la tabla: el principal, los índices, los de campos memos, etc.

Las técnicas de creación de tablas mediante la VCL no resuelven un problema fundamental: la creación de restricciones, principalmente las de integridad referencial, y de valores por omisión. Esta carencia no es grave en el caso de los sistemas SQL, pues la forma preferida de crear tablas en estos sistemas es mediante instrucciones SQL. Pero sí es un problema para Paradox y dBase. Lógicamente, el BDE permite este tipo de operaciones mediante llamadas al API de bajo nivel, pero estas son demasiado complicadas para analizarlas aquí. En el CD que acompaña al libro he incluido un componente que sirve para este propósito. De todos modos, soy partidario de evitar la creación dinámica de tablas en la medida de lo posible.

Validaciones y el Diccionario de Datos

EN EL CAPÍTULO ANTERIOR HEMOS ESTUDIADO LA estructura de los componentes de campos y las propiedades relacionadas con la visualización del contenido de los mismos. La otra cara de la moneda consiste en la validación del contenido de los campos durante la edición. Necesitamos también estudiar la herramienta de ayuda al desarrollo conocida como Diccionario de Datos, que nos permitirá acelerar la configuración de las propiedades de los objetos de acceso a campos.

Validación a nivel de campos

Los componentes de acceso a campos ofrecen dos eventos que se disparan durante el proceso de asignación de valores: *OnValidate* y *OnChange*. El segundo se dispara cuando cambia el contenido del campo, y puede utilizarse para coordinar actualizaciones entre columnas. El más usado, sin embargo, es *OnValidate*, que se emplea para verificar que los valores asignados a un campo sean correctos.

¿Cuándo exactamente se dispara este evento? La respuesta es importante: no debe importarnos cuándo. En realidad, el evento se dispara cuando se va a transferir la información del campo al *buffer* del registro activo. Pero esta operación se lleva a cabo en unas cuantas situaciones diferentes, y es difícil rastrear todos estos casos. No hay que preocuparse de este asunto, pues la VCL se encarga de disparar el evento en el momento adecuado. Por el contrario, debemos estar preparados para dar la respuesta adecuada cuando el hecho suceda. Cuando el sabio señala a la Luna, el tonto solamente ve el dedo.

Sin embargo, es bueno aclarar que *OnValidate* solamente se ejecuta cuando se intentan actualizaciones. Si especificamos una condición de validación sobre una tabla, y ciertos registros ya existentes violan la condición, C++ Builder no protestará al visualizar los campos. Pero si intentamos crear nuevos registros con valores incorrectos, o modificar alguno de los registros incorrectos, se señalará el error.

El siguiente método, programado como respuesta a un evento *OnValidate*, verifica que un nombre propio debe contener solamente caracteres alfabéticos (no somos robots):

```
void __fastcall TmodDatos::VerificarNombrePropio(TField *Sender)
{
    AnsiString S = Sender->AsString;
    for (int i = S.Length(); i > 0; i--)
        if (S[i] != ' ' && !IsCharAlpha(S[i]))
            DatabaseError("Carácter no permitido en nombre propio", 0);
}
```

Este manejador puede ser compartido por varios componentes de acceso a campos, como pueden ser el campo del nombre y el del apellido. Es por esto que se extrae el valor del campo del parámetro *Sender*. Si falla la verificación, la forma de impedir el cambio es interrumpir la operación elevando una excepción; en caso contrario, no se hace nada especial. Para lanzar la excepción he utilizado la función *DatabaseError*. La llamada a esta función es equivalente a la siguiente instrucción:

```
throw EDatabaseError("Carácter no permitido en nombre propio", 0);
```

El problema de utilizar **throw** directamente es que se consume más código, pues también hay que crear el objeto de excepción en línea. La excepción *EDatabaseError*, por convenio, se utiliza para señalar los errores de bases de datos producidos por el usuario o por la VCL, no por el BDE.

Propiedades de validación

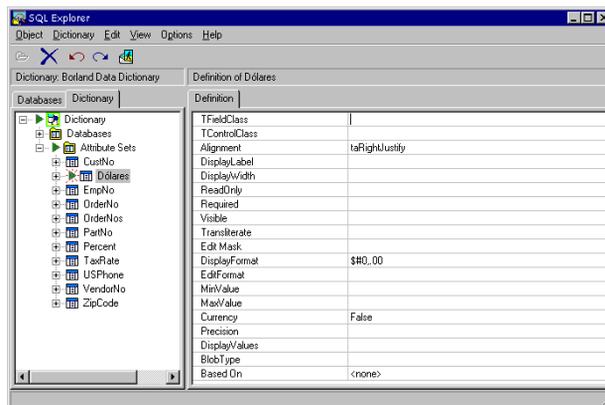
La intercepción del evento *OnValidate* es la forma más general de verificar el contenido de un campo. Pero la mayor parte de las validaciones pueden efectuarse modificando valores de propiedades. Por ejemplo, para comprobar que a una columna no se le asigne un valor nulo se emplea la propiedad *Required* del campo correspondiente. Si el campo es numérico y queremos limitar el rango de valores aceptables, podemos utilizar las propiedades *MinValue* y *MaxValue*. En el caso de los campos alfanuméricos, el formato de la cadena se puede limitar mediante la propiedad *EditMask*, que hemos mencionado anteriormente.

He dejado para más adelante, en este mismo capítulo, el uso de las *restricciones* (*constraints*) y su configuración mediante el Diccionario de Datos.

El Diccionario de Datos

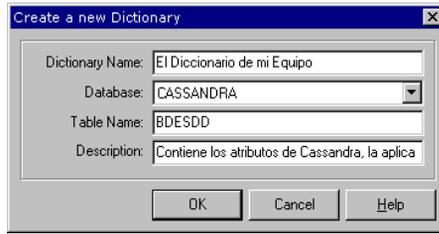
El Diccionario de Datos es una ayuda para el diseño que se administra desde la herramienta *Database Explorer*. El Diccionario nos ayuda a:

- Definir conjuntos de propiedades o *conjuntos de atributos (attribute sets)*, que pueden después asociarse manual o automáticamente a los componentes de acceso a campos que se crean en C++ Builder. Por ejemplo, si nuestra aplicación trabaja con varios campos de porcentajes, puede sernos útil definir que los campos de este tipo se muestren con la propiedad *DisplayFormat* igual a "%#0", y que sus valores oscilen entre 0 y 100.
- Propagar a las aplicaciones clientes restricciones establecidas en el servidor, ya sea a nivel de campos o de tablas, valores por omisión, definiciones de dominios, etcétera.
- Organizar de forma centralizada los criterios de formato y validación entre varios programadores de un mismo equipo, y de una aplicación a otra.



C++ Builder crea al instalarse un Diccionario de Datos por omisión, que se almacena como una tabla Paradox, cuyo nombre por omisión es *bdesdd* y que reside en el alias predefinido *DefaultDD*. Recuerde que esta herramienta es una utilidad que funciona en tiempo de diseño, de modo que la tabla anterior no tiene (no *debe*, más bien) que estar presente junto a nuestro producto final.

Ahora bien, podemos trabajar con otro diccionario, que puede estar almacenado en cualquier otro formato de los reconocidos por el BDE. Este diccionario puede incluso almacenarse en un servidor SQL y ser compartido por todos los miembros de un equipo de programación. Mediante el comando de menú *Dictionary | New* uno de los programadores del equipo, digamos que usted, crea una nueva tabla para un nuevo diccionario, indicando el alias donde residirá, el nombre de la tabla y una descripción para el diccionario:



Después, mediante el comando de menú *Dictionary | Register* cualquier miembro del equipo, digamos que yo, puede registrar ese diccionario desde *otra* máquina, para utilizarlo con *su* C++ Builder. También es útil el comando *Dictionary | Select*, para activar alguno de los diccionarios registrados en determinado ordenador.

Conjuntos de atributos

Son dos los objetos almacenados en un Diccionario de Datos: los conjuntos de atributos e información sobre bases de datos completas. Un conjunto de atributos indica valores para propiedades comunes de campos. Existen dos formas de crear esta definición: guardando los atributos asociados a un campo determinado, o introduciéndolos directamente en el Diccionario de Datos. Para salvar las modificaciones realizadas a un campo desde C++ Builder, hay que seleccionarlo en el Editor de Campos, pulsar el botón derecho del ratón y ejecutar el comando *Save attributes as*, que nos pedirá un nombre para el conjunto de atributos.

La otra vía es crear directamente el conjunto de atributos en el propio Diccionario. Digamos que nuestra aplicación debe trabajar con precios en dólares, mientras que nuestra moneda local es diferente. Nos situamos entonces sobre el nodo *Attribute sets* y ejecutamos *Object | New*. Renombramos el nuevo nodo como *Dollar*, y modificamos las siguientes propiedades, en el panel de la derecha:

Propiedad	Valor
<i>Currency</i>	<i>False</i>
<i>DisplayFormat</i>	<i>\$\$#0,.00</i>

¿Qué se puede hacer con este conjunto de atributos, una vez creado? Asociarlo a campos, por supuesto. Nuestra hipotética aplicación maneja una tabla *Articulos* con un campo *PrecioDolares*, que ha sido definido con el tipo *money* de Paradox o de MS SQL Server. C++ Builder, por omisión, trae un campo de tipo *TCurrencyField*, cuya propiedad *Currency* aparece como *True*. El resultado: nuestros dólares se transforman mágicamente en pesetas (y pierden todo su valor). Pero seleccionamos el menú local del campo, y ejecutamos el comando *Associate attributes*, seleccionando el conjunto de atributos *Dollar* definido hace poco. Entonces C++ Builder lee los valores de las

propiedades *Currency* y *DisplayFormat* desde el Diccionario de Datos y los copia en las propiedades del campo deseado. Y todo vuelve a la normalidad.

Dos de los atributos más importantes que se pueden definir en el Diccionario son *TFieldClass* y *TControlClass*. Mediante el primero podemos establecer explícitamente qué tipo de objeto de acceso queremos asociar a determinado campo; esto es útil sobre todo con campos de tipo BLOB. *TControlClass*, por su parte, determina qué tipo de control debe crearse cuando se arrastra y suelta un componente de campo sobre un formulario en tiempo de diseño.

Si un conjunto de atributos se modifica después de haber sido asociado a un campo, los cambios no se propagarán automáticamente a las propiedades de dicho campo. Habrá entonces que ejecutar el comando del menú local *Retrieve attributes*.

Importando bases de datos

Pero si nos limitamos a las técnicas descritas en la sección anterior, tendremos que configurar atributos campo por campo. Una alternativa consiste en *importar* el esquema de la base de datos dentro del Diccionario, mediante el comando de menú *Dictionary | Import from database*. Aparece un cuadro de diálogo para que seleccionemos uno de los alias disponibles; nos debemos armar de paciencia, porque la operación puede tardar un poco.

Muchas aplicaciones trabajan con un alias de sesión, definido mediante un componente *TDatabase* que se sitúa dentro de un módulo de datos, en vez de utilizar alias persistentes. Si ejecutamos Database Explorer como una aplicación independiente no podremos importar esa base de datos, al no estar disponible el alias en cuestión. La solución es invocar al Database Explorer desde el Entorno de Desarrollo, con la aplicación cargada y la base de datos conectada. Todos los alias de sesión activos en la aplicación podrán utilizarse entonces desde esta utilidad.

Cuando ha finalizado la importación, aparece un nuevo nodo para nuestra base de datos bajo el nodo *Databases*. El nuevo nodo contiene todas las tablas y los campos existentes en la base de datos, y a cada campo se le asocia automáticamente un conjunto de atributos. Por omisión, el Diccionario crea un conjunto de atributos por cada campo que tenga propiedades dignas de mención: una restricción (espere a la próxima sección), un valor por omisión, etc. Esto es demasiado, por supuesto. Después de la importación, debemos sacar factor común de los conjuntos de atributos similares y asociarlos adecuadamente a los campos. La labor se facilita en InterBase si

hemos creado *dominios*, como explicamos en el capítulo 9. Supongamos que definimos el dominio *Dollar* en la base de datos mediante la siguiente instrucción:

```
create domain Dollar as  
    numeric(15, 2) default 0 not null;
```

A partir de esta definición podremos definir columnas de tablas cuyo tipo de datos sea *Dollar*. Entonces el Diccionario de Datos, al importar la base de datos, creará automáticamente un conjunto de atributos denominado *Dollar*, y asociará correctamente los campos que pertenecen a ese conjunto.

¿Para qué todo este trabajo? Ahora, cuando traigamos una tabla a la aplicación y utilicemos el comando *Add fields* para crear objetos persistentes de campos, C++ Builder podrá asignar de forma automática los conjuntos de atributos a estos campos.

Los tipos definidos por el usuario de MS SQL Server también son utilizados por el Diccionario para identificar conjuntos de atributos durante la importación de tablas.

Evaluando restricciones en el cliente

Los sistemas de bases de datos cliente/servidor, y algunas bases de datos de escritorio, permiten definir restricciones en las tablas que, normalmente, son verificadas por el propio servidor de datos. Por ejemplo, que el nombre del cliente no puede estar vacío, que la fecha de venta debe ser igual o anterior a la fecha de envío de un pedido, que un descuento debe ser mayor que cero, pero menor que cien...

Un día de verano, un usuario de nuestra aplicación (¡ah, los usuarios!) se enfrasca en rellenar un pedido, y vende un raro disco de Hendrix con un descuento del 125%. Al enviar los datos al servidor, éste detecta el error y notifica a la aplicación acerca del mismo. Un registro pasó a través de la red, un error nos fue devuelto; al parecer, poca cosa. Pero multiplique este tráfico por cincuenta puestos de venta, y lo poco se convierte en mucho. Además, ¿por qué hacer esperar tanto al usuario para preguntarle si es tonto, o si lo ha hecho a posta? Este tipo de validación sencilla puede ser ejecutada perfectamente por nuestra aplicación, pues solamente afecta a columnas del registro activo. Otra cosa muy diferente sería, claro está, intentar verificar por duplicado en el cliente una restricción de unicidad.

En C++ Builder 3 se introdujeron cuatro nuevas propiedades, de tipo *AnsiString*, para los componentes de acceso a campos:

Propiedad	Significado
<i>DefaultExpression</i>	Valor por omisión del campo
<i>CustomConstraint</i>	Restricciones importadas desde el servidor
<i>ImportedConstraint</i>	Restricciones adicionales impuestas en el cliente
<i>ConstraintErrorMessage</i>	Mensaje de error cuando se violan restricciones

DefaultExpression es una expresión SQL que sirve para inicializar un campo durante las inserciones. No puede contener nombres de campos. Antes de la versión 3, la inicialización de los valores de los campos debía realizarse programando una respuesta al evento *OnNewRecord* del conjunto de datos al que pertenecía. Si un campo debe inicializarse con un valor constante, es más cómodo utilizar *DefaultExpression*.

Hay un pequeño *bug* en la VCL: cuando se mezclan inicializaciones en el evento *OnNewRecord* con valores en las propiedades *DefaultExpression*, se producen comportamientos anómalos. Evite las mezclas, que no son buenas para la salud.

Podemos asignar directamente un valor constante en *DefaultExpression* ... pero si hemos asociado un conjunto de atributos al campo, C++ Builder puede leer automáticamente el valor por omisión asociado y asignarlo. Este conjunto de atributos puede haber sido configurado de forma manual, pero también puede haberse creado al importar la base de datos dentro del Diccionario. En este último caso, el Diccionario de Datos ha actuado como eslabón intermedio en la propagación de reglas de empresa desde el servidor al cliente:



Lo mismo se aplica a la propiedad *ImportedConstraint*. Esta propiedad recibe su valor desde el Diccionario de Datos, y debe contener una expresión SQL evaluable en SQL Local; léase, en el dialecto de Paradox y dBase. ¿Por qué permitir que esta propiedad pueda modificarse en tiempo de diseño? Precisamente porque la expresión importada puede ser incorrecta en el dialecto local. En ese caso, podemos eliminar toda o parte de la expresión. Normalmente, el Diccionario de Datos extrae las restricciones para los conjuntos de atributos de las cláusulas **check** de SQL definidas a nivel de columna.

Si, por el contrario, lo que se desea es añadir nuevas restricciones, debemos asignarlas en la propiedad *CustomConstraint*. Se puede suministrar cualquier expresión lógica de SQL Local, y para representar el valor actual del campo hay que utilizar un identifi-

gador cualquiera que no sea utilizado por SQL. Por ejemplo, esta puede ser una expresión aplicada sobre un campo de tipo cadena de caracteres:

```
x <> '' and x not like '% %'
```

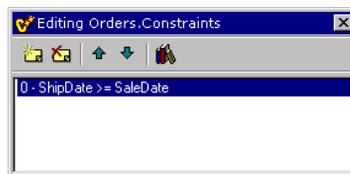
Esta expresión verifica que el campo no esté vacío y que no contenga espacios en blanco.

Cuando se viola cualquiera de las restricciones anteriores, *ImportedConstraint* ó *CustomConstraint*, por omisión se muestra una excepción con el mensaje “Record or field constraint failed”, más la expresión que ha fallado en la segunda línea del mensaje. Si queremos mostrar un mensaje personalizado, debemos asignarlo a la propiedad *ConstraintErrorMessage*.

Sin embargo, no todas las restricciones **check** se definen a nivel de columna, sino que algunas se crean a nivel de tabla, casi siempre cuando involucran a dos o más campos a la vez. Por ejemplo:

```
create table Pedidos (
    /* Restricción a nivel de columna */
    FormaPago varchar(10)
        check (FormaPago in ('EFECTIVO', 'TARJETA')),
    /* Restricción a nivel de tabla */
    FechaVenta date not null,
    FechaEnvio date,
    check (FechaVenta <= FechaEnvio),
    /* ... */
);
```

Las restricciones a nivel de tabla se propagan a una propiedad de *TTable* denominada *Constraints*, que contiene tanto las restricciones importadas como alguna restricción personalizada añadida por el programador.



La imagen anterior muestra el editor de la propiedad *Constraints* de C++ Builder 4. El botón de la derecha permite leer las restricciones encontradas durante la importación de la base de datos al Diccionario; esta vez, C++ Builder no lo hace por sí mismo. Mediante el botón de la derecha se añaden restricciones personalizadas a la tabla. Cada restricción a nivel de tabla, venga de donde venga, tiene su propia propiedad *ConstraintErrorMessage*.

Controles de datos y fuentes de datos

ESTE CAPÍTULO TRATA ACERCA DE LOS CONTROLES que ofrece la VCL para visualizar y editar información procedente de bases de datos, la filosofía general en que se apoyan y las particularidades de cada uno de ellos. Es el momento de ampliar, además, nuestros conocimientos acerca de un componente esencial para la sincronización de estos controles, el componente *TDataSource*. Veremos cómo un control “normal” puede convertirse, gracias a una sencilla técnica basada en este componente, en un flamante control de acceso a datos. Por último, estudiaremos cómo manejar campos BLOB, que pueden contener grandes volúmenes de información, dejando su interpretación a nuestro cargo.

Controles *data-aware*

Los controles de bases de datos son conocidos en la jerga de C++ Builder como controles *data-aware*. Estos controles son, generalmente, versiones especializadas de controles “normales”. Por ejemplo, *TDBMemo* es una versión orientada a bases de datos del conocido *TMemo*. Al tener por separado los controles de bases de datos y los controles tradicionales, C++ Builder evita que una aplicación que no haga uso de bases de datos¹³ tenga que cargar con todo el código necesario para estas operaciones. No sucede así con Visual Basic, lenguaje en que todos los controles pueden potencialmente conectarse a una base de datos.



Los controles de acceso a datos de C++ Builder se pueden dividir en dos grandes grupos:

¹³ Existen.

- Controles asociados a campos
- Controles asociados a conjuntos de datos

Los controles asociados a campos visualizan y editan una columna particular de una tabla. Los componentes *TDBEdit* (cuadros de edición) y *TDBImage* (imágenes almacenadas en campos gráficos) pertenecen a este tipo de controles. Los controles asociados a conjuntos de datos, en cambio, trabajan con la tabla o consulta como un todo. Las rejillas de datos y las barras de navegación son los ejemplos más conocidos de este segundo grupo. Todos los controles de acceso a datos orientados a campos tienen un par de propiedades para indicar con qué datos trabajan: *DataSource* y *DataField*; estas propiedades pueden utilizarse en tiempo de diseño. Por el contrario, los controles orientados al conjunto de datos solamente disponen de la propiedad *DataSource*. La conexión con la fuente de datos es fundamental, pues es este componente quien notifica al control de datos de cualquier alteración en la información que debe visualizar.

Casi todos los controles de datos permiten, de una forma u otra, la edición de su contenido. En el capítulo 17 sobre conjuntos de datos explicamos que hace falta que la tabla esté en uno de los modos *dsEdit* ó *dsInsert* para poder modificar el contenido de un campo. Pues bien, todos los controles de datos son capaces de colocar a la tabla asociada en este modo, de forma automática, cuando se realizan modificaciones en su interior. Este comportamiento se puede modificar con la propiedad *AutoEdit* del *data source* al que se conectan. Cada control, además, dispone de una propiedad *ReadOnly*, que permite utilizar el control únicamente para visualización.

Actualmente, los controles de datos de C++ Builder son los siguientes:

Nombre de la clase	Explicación
Controles orientados a campos	
<i>TDBText</i>	Textos no modificables (no consumen recursos)
<i>TDBEdit</i>	Cuadros de edición
<i>TDBMemo</i>	Textos sin formato con múltiples líneas
<i>TDBImage</i>	Imágenes BMP y WMF
<i>TDBListBox</i>	Cuadros de lista (contenido fijo)
<i>TDBComboBox</i>	Cuadros de combinación (contenido fijo)
<i>TDBCheckBox</i>	Casillas de verificación (dos estados)
<i>TDBRadioGroup</i>	Grupos de botones (varios valores fijos)
<i>TDBLookupListBox</i>	Valores extraídos desde otra tabla asociada
<i>TDBLookupComboBox</i>	Valores extraídos desde otra tabla asociada
<i>TDBRichEdit</i>	Textos en formato RTF

Nombre de la clase	Explicación
Controles orientados a conjuntos de datos	
<i>TDBGrid</i>	Rejillas de datos para la exploración
<i>TDBNavigator</i>	Control de navegación y estado
<i>TDBCtrlGrid</i>	Rejilla que permite incluir controles

En este capítulo nos ocuparemos principalmente de los controles orientados a campos. El resto de los controles serán estudiados en el siguiente capítulo.

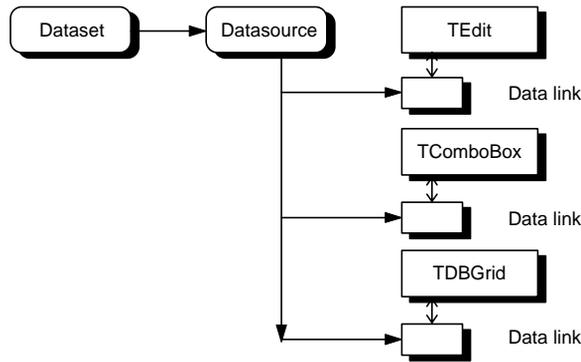
He omitido intencionalmente el control *TDBChart* de la lista anterior, por trabajar con un sistema distinto a los demás. Este componente se estudiará más adelante.

Los enlaces de datos

Según lo explicado, todos los controles de datos deben ser capaces de reconocer y participar en el juego de las notificaciones, y esto supone la existencia de un montón de código común a todos ellos. Pero, si observamos el diagrama de herencia de la VCL, notaremos que no existe un ancestro compartido propio para los controles *data-aware*. ¿Qué solución se ha utilizado en la VCL para evitar la duplicación de código?

La respuesta la tiene un objeto generalmente ignorado por el programador de C++ Builder: *TDataLink*, y su descendiente *TFieldDataLink*. Este desconocimiento es comprensible, pues no es un componente visual, y sólo es imprescindible para el desarrollador de componentes. Cada control de datos crea durante su inicialización un componente interno perteneciente a una de estas clases. Es este componente interno el que se conecta a la fuente de datos, y es también a éste a quien la fuente de datos envía las notificaciones acerca del movimiento y modificaciones que ocurren en el conjunto de datos subyacente. Todo el tratamiento de las notificaciones se produce entonces, al menos de forma inicial, en el *data link*. Esta técnica es conocida como *delegación*, y nos evita el uso de la herencia múltiple, recurso no soportado por la VCL hasta el momento.

El siguiente esquema muestra la relación entre los componentes de acceso y edición de datos:



Creación de controles de datos

Podemos crear controles de datos en un formulario trayendo uno a uno los componentes deseados desde la Paleta e inicializando sus propiedades *DataSource* y *DataField*. Esta es una tarea tediosa. Muchos programadores utilizaban en las primeras versiones de C++ Builder el Experto de Datos (*Database form expert*) para crear un formulario con los controles deseados, y luego modificar este diseño inicial. Este experto, sin embargo, tiene un par de limitaciones importantes: en primer lugar, trabaja con un tamaño fijo de la ventana, lo cual nos obliga a realizar desplazamientos cuando no hay espacio para los controles, aún cuando aumentando el tamaño de la ventana se pudiera resolver el problema. El otro inconveniente es que siempre genera un nuevo componente *TTable* ó *TQuery*, no permitiendo utilizar componentes existentes de estos tipos. Esto es un problema, pues lo usual es definir primero los conjuntos de datos en un módulo aparte, para poder programar reglas de empresa de forma centralizada.

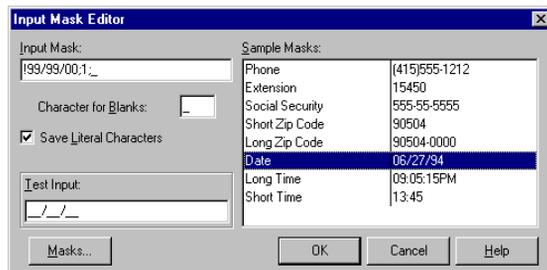
Podemos arrastrar campos desde el Editor de Campos sobre un formulario. Cuando lo hacemos, se crea automáticamente un control de datos asociado al campo. Junto con el control, se crea también una etiqueta, de clase *TLabel*, con el título extraído de la propiedad *DisplayLabel* del campo. Recuerde que esta propiedad coincide inicialmente con el nombre del campo, de modo que si las columnas de sus tablas tienen nombre crípticos como *NomCli*, es conveniente modificar primero las etiquetas de visualización en los campos antes de crear los controles. Adicionalmente, C++ Builder asigna a la propiedad *FocusControl* de los componentes *TLabel* creados el puntero al control asociado. De este modo, si la etiqueta tiene un carácter subrayado, podemos hacer uso de este carácter como abreviatura para la selección del control.

En cuanto al tipo de control creado, C++ Builder tiene sus reglas por omisión: casi siempre se crea un *TDBEdit*. Si el campo es un campo de búsqueda, crea un componente *TDBLookupComboBox*. Si es un memo o un campo gráfico, se crea un control *TDBMemo* o un *TDBImage*. Si el campo es lógico, se crea un *TDBCheckBox*. Estas

reglas implícitas, sin embargo, pueden modificarse por medio del Diccionario de Datos. Si el campo que se arrastra tiene definido un conjunto de atributos, el control a crear se determina por el valor almacenado en la propiedad *TControlClass* del conjunto de atributos en el Diccionario de Datos.

Los cuadros de edición

La forma más general de editar un campo simple es utilizar un control *TDBEdit*. Este componente puede utilizarse con campos de cadenas de caracteres, numéricos, de fecha y de cualquier tipo en general que permita conversiones desde y hacia cadenas de caracteres. Los cuadros de edición con conexión a bases de datos se derivan de la clase *TCustomMaskEdit*. Esto es así para poder utilizar la propiedad *EditMask*, perteneciente a la clase *TField*, durante la edición de un campo. Sin embargo, *EditMask* es una propiedad protegida de *TDBEdit*; el motivo es permitir que la máscara de edición se asigne directamente desde el campo asociado, dentro de la VCL, y que el programador no pueda interferir en esta asignación. Si el campo tiene una máscara de edición definida, la propiedad *IsMasked* del cuadro de edición se vuelve verdadera.



Windows no permite definir alineación para el texto de un cuadro de edición, a no ser que el estilo del cuadro sea multilíneas. Si nos fijamos un poco, el control *TEdit* estándar no tiene una propiedad *Alignment*. Sin embargo, es común mostrar los campos numéricos de una tabla justificados a la derecha. Es por eso que, en el código fuente de la VCL, se realiza un truco para permitir los distintos tipos de alineación en los componentes *TDBEdit*. Es importante comprender que *Alignment* solamente funciona durante la visualización, no durante la edición. La alineación se extrae de la propiedad correspondiente del componente de acceso al campo.

Los eventos de este control que utilizaremos con mayor frecuencia son *OnChange*, *OnKeyPress*, *OnKeyDown* y *OnExit*. *OnChange* se produce cada vez que el texto en el control es modificado. Puede causar confusión el que los componentes de campos tengan un evento también nombrado *OnChange*. Este último evento se dispara cuando se modifica el contenido del campo, lo cual sucede cuando se realiza una asignación al componente de campo. Si el campo se está editando en un *TDBEdit*, esto sucede al abandonar el control, o al pulsar INTRO estando el control seleccio-

nado. En cambio, el evento *OnChange* del control de edición se dispara cada vez que se cambia algo en el control. El siguiente evento muestra cómo pasar al control siguiente cuando se alcanza la longitud máxima admitida por el editor. Este comportamiento era frecuente en programas realizados para MS-DOS:

```
void __fastcall TForm1::DBEdit1Change(TObject *Sender)
{
    if (Visible)
    {
        TDBEdit &edit = dynamic_cast<TDBEdit*>(*Sender);
        if (edit.Text.Length() >= edit.MaxLength)
            SelectNext(&edit, True, True);
    }
}
```

Hasta la versión 3, C++ Builder arrastró un pequeño problema en relación con los cuadros de edición asociados a un campo numérico: la propiedad *MaxLength* siempre se hacía 0 cuando se crea el control. Aunque asignásemos algo a esta propiedad en tiempo de diseño, siempre se perdía en tiempo de ejecución. En la versión 4 se ha corregido el error.

Editores de texto

Cuando el campo a editar contiene varias líneas de texto, debemos utilizar un componente *TDBMemo*. Si queremos además que el texto tenga formato y permita el uso de negritas, cursivas, diferentes colores y alineaciones podemos utilizar el componente *TDBRichEdit*.

TDBMemo está limitado a un máximo de 32KB de texto, además de permitir un solo tipo de letra y de alineación para todo su contenido. Las siguientes propiedades han sido heredadas del componente *TMemo* y determinan la apariencia y forma de interacción con el usuario:

Propiedad	Significado
<i>Alignment</i>	La alineación del texto dentro del control.
<i>Lines</i>	El contenido del control, como una lista de cadenas de caracteres.
<i>ScrollBars</i>	Determina qué barras de desplazamiento se muestran.
<i>WantTabs</i>	Si está activa, el editor interpreta las tabulaciones como tales; en caso contrario, sirven para seleccionar el próximo control.
<i>WordWrap</i>	Las líneas pueden dividirse si sobrepasan el extremo derecho del editor.

La propiedad *AutoDisplay* es específica de este tipo de controles. Como la carga y visualización de un texto con múltiples líneas puede consumir bastante tiempo, se

puede asignar *False* a esta propiedad para que el control aparezca vacío al mover la fila activa. Luego se puede cargar el contenido en el control pulsando INTRO sobre el mismo, o llamando al método *LoadMemo*.

El componente *TDBRichEdit*, por su parte, es similar a *TDBMemo*, excepto por la mayor cantidad de eventos y las propiedades de formato.

Textos no editables

El tipo *TLabel* tiene un equivalente *data-aware*: el tipo *TDBText*. Mediante este componente, se puede mostrar información como textos no editables. Gracias a que este control desciende por herencia de la clase *TGraphicControl*, desde el punto de vista de Windows no es una ventana y no consume recursos. Si utilizamos un *TDBEdit* con la propiedad *ReadOnly* igual a *True*, consumiremos un *handle* de ventana. En compensación, con el *TDBEdit* podemos seleccionar texto y copiarlo al Portapapeles, cosa imposible de realizar con un *TDBText*.

Además de las propiedades usuales en los controles de bases de datos, *DataSource* y *DataField*, la otra propiedad interesante es *AutoSize*, que indica si el ancho del control se ajusta al tamaño del texto o si se muestra el texto en un área fija, truncándolo si la sobrepasa.

Combos y listas con contenido fijo

Los componentes *TDBComboBox* y *TDBListBox* son las versiones *data-aware* de *TComboBox* y *TListBox*, respectivamente. Se utilizan, preferentemente, cuando hay un número bien determinado de valores que puede aceptar un campo, y queremos ayudar al usuario para que no tenga que teclear el valor, sino que pueda seleccionarlo con el ratón o el teclado. En ambos casos, la lista de valores predeterminados se indica en la propiedad *Items*, de tipo *TStrings*.

De estos dos componentes, el cuadro de lista es el menos utilizado. No permite introducir valores diferentes a los especificados; si el campo asociado del registro activo tiene ya un valor no especificado, no se selecciona ninguna de las líneas. Tampoco permite búsquedas incrementales sobre listas ordenadas. Si las opciones posibles en el campo son pocas, la mayoría de los usuarios y programadores prefieren utilizar el componente *TDBRadioGroup*, que estudiaremos en breve.

En cambio, *TDBComboBox* es más flexible. En primer lugar, nos deja utilizar tres estilos diferentes de interacción mediante la propiedad *Style*; en realidad son cinco estilos, pero dos de ellos tienen que ver más con la apariencia del control que con la interfaz con el usuario:

Estilo	Significado
<i>csSimple</i>	La lista siempre está desplegada. Se pueden teclear valores que no se encuentran en la lista.
<i>csDropDown</i>	La lista está inicialmente recogida. Se pueden teclear valores que no se encuentran en la lista.
<i>csDropDownList</i>	La lista está inicialmente recogida. No se pueden teclear valores que no se encuentren en la lista.
<i>csOwnerDrawFixed</i>	El contenido de la lista lo dibuja el programador. Líneas de igual altura.
<i>csOwnerDrawVariable</i>	El contenido de la lista lo dibuja el programador. La altura de las líneas la determina el programador.

Por otra parte, la propiedad *Sorted* permite ordenar dinámicamente los elementos de la lista desplegable de los combos. Los combos con el valor *csDropDownList* en la propiedad *Style*, y cuya propiedad *Sorted* es igual a *True*, permiten realizar búsquedas incrementales. Si, por ejemplo, un combo está mostrando nombres de países, al teclear la letra *A* nos situaremos en Abisinia, luego una *N* nos llevará hasta la Nantártida, y así en lo adelante. Si queremos iniciar la búsqueda desde el principio, o sea, que la *N* nos sitúe sobre Nepal, podemos pulsar ESC o RETROCESO ... o esperar dos segundos. Esto último es curioso, pues la duración de ese intervalo está incrustada en el código de la VCL y no puede modificarse fácilmente.

Cuando el estilo de un combo es *csOwnerDrawFixed* ó *csOwnerDrawVariable*, es posible dibujar el contenido de la lista desplegable; para los cuadros de lista, *Style* debe valer *lbOwnerDrawFixed* ó *lbOwnerDrawVariable*. Si utilizamos alguno de estos estilos, tenemos que crear una respuesta al evento *OnDrawItem* y, si el estilo de dibujo es con alturas variables, el evento *OnMeasureItem*. Estos eventos tienen los siguientes parámetros:

```
void __fastcall TForm1::DBComboBox1DrawItem(TWinControl *Control,
      int Index, TRect Rect, TOwnerDrawState State)
{
}

void __fastcall TForm1::DBComboBox1MeasureItem(TWinControl *Control,
      int Index, int &Altura)
{
}
```

Para ilustrar el uso de estos eventos, crearemos un combo que pueda seleccionar de una lista de países, con el detalle de que cada país aparezca representado por su bandera (o por lo que más le apetezca dibujar). Vamos a inicializar el combo en el evento de creación de la ventana, y para complicar un poco el código leeremos los mapas de bits necesarios desde una tabla de países:

```

void __fastcall TForm1::FormCreate(TObject *Sender)
{
    std::auto_ptr<TTable> table(new TTable(NULL));
    table->DatabaseName = "Prueba";
    table->TableName = "Paises.DB";
    table->Open();
    while (! table->Eof)
    {
        Graphics::TBitmap *Bmp = new Graphics::TBitmap;
        try
        {
            Bmp->Assign(table->FieldByName("Bandera"));
            DBComboBox1->Items->AddObject(
                table->FieldByName("Pais")->AsString, Bmp);
        }
        catch(Exception&)
        {
            delete Bmp;
            throw;
        }
        table->Next();
    }
}
    
```

El código que dibuja el mapa de bits al lado del texto es el siguiente:

```

void __fastcall TForm1::DBComboBox1DrawItem(TWinControl *Control,
int Index, TRect Rect, TOwnerDrawState State)
{
    TDBComboBox &c = dynamic_cast<TDBComboBox&>(*Control);
    int ih = c.ItemHeight;
    c.Canvas->FillRect(Rect);
    c.Canvas->StretchDraw(
        Bounds(Rect.Left + 2, Rect.Top, ih, ih),
        (TBitmap*)(c.Items->Objects[Index]));
    TextOut(Rect.Left + ih + 6, Rect.Top,
        c.Items->Strings[Index]);
}
    
```



Una propiedad poco conocida de *TDBComboBox*, que éste hereda de *TComboBox*, es *DroppedDown*, de tipo lógico. *DroppedDown* es una propiedad de tiempo de ejecución, y permite saber si la lista está desplegada o no. Pero también permite asignaciones, para ocultar o desplegar la lista. Si el usuario quiere que la tecla + del teclado numérico despliegue todos los combos de una ficha, puede asignar *True* a la propiedad *KeyPreview* del formulario y crear la siguiente respuesta al evento *OnKeyDown*:

```

void __fastcall TForm1::FormKeyDown(TObject *Sender;
    Word &Key, TShiftState Shift)
{
    if (Key == VK_ADD &&
        dynamic_cast<TCustomComboBox*>(ActiveControl))
    {
        ((TCustomComboBox*) ActiveControl)->DroppedDown = True;
        Key = 0;
    }
}

```

Combos y listas de búsqueda

En cualquier diseño de bases de datos abundan los campos sobre los que se han definido restricciones de integridad referencial. Estos campos hacen referencia a valores almacenados como claves primarias de otras tablas. Pero casi siempre estos enlaces se realizan mediante claves artificiales, como es el caso de los códigos. ¿Qué me importa a mí que la Coca-Cola sea el artículo de código 4897 de mi base de datos particular? Soy un enemigo declarado de los códigos: en la mayoría de los casos se utilizan para permitir relaciones más eficientes entre tablas, por lo que deben ser internos a la aplicación. El usuario, en mi opinión, debe trabajar lo menos posible con códigos. ¿Qué es preferible, dejar que el usuario teclee cuatro dígitos, 4897, o que teclee el prefijo del nombre del producto?

Por estas razones, cuando tenemos que editar un campo de este tipo es preferible utilizar la clave verdadera a la clave artificial. En el caso del artículo, es mejor que el usuario pueda seleccionar el nombre del artículo que obligarle a introducir un código. Esta traducción, de código a descripción, puede efectuarse a la hora de visualizar mediante los campos de búsqueda, que ya hemos estudiado. Estos campos, no obstante, son sólo para lectura; si queremos editar el campo original, debemos utilizar los controles *TDBLookupListBox* y *TDBLookupComboBox*.

Las siguientes propiedades son comunes a las listas y combos de búsqueda, y determinan el algoritmo de traducción de código a descripción:

Propiedad	Significado
<i>DataSource</i>	La fuente de datos original. Es la que se modifica.
<i>DataField</i>	El campo original. Es el campo que contiene la referencia.
<i>ListSource</i>	La fuente de datos a la que se hace referencia.
<i>KeyField</i>	El campo al que se hace referencia.
<i>ListField</i>	Los campos que se muestran como descripción.

Cuando se arrastra un campo de búsqueda desde el Editor de Campos hasta la superficie de un formulario, el componente creado es de tipo *TDBLookupComboBox*. En este caso especial, solamente hace falta dar el nombre del campo de búsqueda en la

propiedad *DataField* del combo o la rejilla, pues el resto del algoritmo de búsqueda es deducible a partir de las propiedades del campo base.

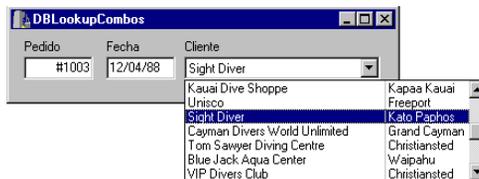
Los combos de búsquedas funcionan con el estilo equivalente al de un *TDBCCombo-Box* ordenado y con el estilo *csDropDownList*. Esto quiere decir que no se pueden introducir valores que no se encuentren en la tabla de referencia. Pero también significa que podemos utilizar las búsquedas incrementales por teclado. Y esto es también válido para los componentes *TDBLookupListBox*.

Más adelante, cuando estudiemos la comunicación entre el BDE y las bases de datos cliente/servidor, veremos que la búsqueda incremental en combos de búsqueda es una característica *mu*y peligrosa. La forma en que el BDE implementa por omisión las búsquedas incrementales insensibles a mayúsculas es un despilfarrero. ¿Una solución? Utilice ventanas emergentes para la selección de valores, implementando usted mismo el mecanismo de búsqueda incremental, o diseñe su propio combo de búsqueda. Recuerde que esta advertencia solamente vale para bases de datos cliente/servidor.

Tanto para el combo como para las listas pueden especificarse varios campos en *ListField*; en este caso, los nombres de campos se deben separar por puntos y comas:

```
DBLookupListBox1->ListField = "Nombre;Apellidos";
```

Si son varios los campos a visualizar, en el cuadro de lista de *TDBLookupListBox*, y en la lista desplegable de *TDBLookupComboBox* se muestran en columnas separadas. En el caso del combo, en el cuadro de edición solamente se muestra uno de los campos: aquel cuya posición está indicada por la propiedad *ListFieldIndex*. Como por omisión esta propiedad vale 0, inicialmente se muestra el primer campo de la lista. *ListFieldIndex* determina, en cualquier caso, cuál de los campos se utiliza para realizar la búsqueda incremental en el control.



El combo de la figura anterior ha sido modificado de forma tal que su lista desplegable tenga el ancho suficiente para mostrar las dos columnas especificadas en *ListField*. La propiedad *DropDownWidth*, que por omisión vale 0, indica el ancho en píxeles de la lista desplegable. Por otra parte, *DropDownRows* almacena el número de filas que se despliegan a la vez, y *DropDownAlign* es la alineación a utilizar.

Esencia y apariencia

Con mucha frecuencia me hacen la siguiente pregunta: ¿cómo puedo inicializar un combo de búsqueda para que no aparezca nada en él (o para que aparezca determinado valor)? Desde mi punto de vista, esta pregunta revela el mismo problema que la siguiente: ¿cómo muevo una imagen al control *TDBImage*?

Hay un defecto de razonamiento en las dos preguntas anteriores. El programador *ve* el combo, y quiere cambiar los datos *en el combo*. Pero pierde de vista que el combo es una *manifestación* del campo subyacente. Hay que buscar la esencia, no la apariencia. Así que cuando desee eliminar el contenido de un *TDBLookupComboBox* lo que debe hacer es asignar el valor nulo al campo asociado.

Lo mismo sucederá, como veremos al final del capítulo, cuando deseemos almacenar una imagen en un *TDBImage*: sencillamente cargaremos esta imagen en el campo de la tabla o consulta. Y no crea que estoy hablando de errores de programación infrecuentes: he visto programas que para mover una imagen a un campo la copian primero en el Portapapeles, fuerzan a un *TDBImage* a que pegue los datos y luego utilizan *Post* para hacer permanente la modificación. Es evidente que el contenido del Portapapeles no debe ser modificado a espaldas del usuario, por lo que se trata de una técnica pésima. Más adelante veremos cómo utilizar campos *blob* para este propósito.

Casillas de verificación y grupos de botones

Si un campo admite solamente dos valores, como en el caso de los campos lógicos, es posible utilizar para su edición el componente *TDBCheckBox*. Este componente es muy sencillo, y su modo de trabajo está determinado por el tipo del campo asociado. Si el campo es de tipo *TBooleanField*, el componente traduce el valor *True* como casilla marcada, y *False* como casilla sin marcar. En ciertos casos, conviene utilizar la propiedad *AllowGrayed*, que permite que la casilla tenga tres estados; el tercer estado, la casilla en color gris, se asocia con un valor nulo en el campo.

Si el campo no es de tipo lógico, las propiedades *ValueChecked* y *ValueUnchecked* determinan las cadenas equivalentes a la casilla marcada y sin marcar. En estas propiedades se pueden almacenar varias cadenas separadas por puntos y comas. De este modo, el componente puede aceptar varias versiones de lo que la tabla considera valores “marcados” y “no marcados”:

```
DBCheckBox1->ValueChecked = "Sí;Yes;Oui;Bai";
DBCheckBox1->ValueUnchecked = "No;Ez";
```

Por su parte, *TDBRadioGroup* es una versión orientada a bases de datos del componente estándar *TRadioGroup*. Puede utilizarse como alternativa al cuadro de lista con contenido fijo cuando los valores que podemos utilizar son pocos y se pueden mostrar en pantalla todos a la vez. La propiedad *Items*, de tipo *TStrings*, determina los textos que se muestran en pantalla. Si está asignada la propiedad *Values*, indica qué cadenas almacenaremos en el campo para cada opción. Si no hemos especificado algo en esta propiedad, se almacenan las mismas cadenas asignadas en *Items*.

Imágenes extraídas de bases de datos

En un campo BLOB de una tabla se pueden almacenar imágenes en los más diversos formatos. Si el formato de estas imágenes es el de mapas de bits o de metaficheros, podemos utilizar el componente *TDBImage*, que se basa en el control *TImage*, para visualizar y editar estos datos. Las siguientes propiedades determinan la apariencia de la imagen dentro del control:

Propiedad	Significado
<i>Center</i>	La imagen aparece centrada o en la esquina superior izquierda
<i>Stretch</i>	Cuando es verdadera, la imagen se expande o contrae para adaptarse al área del control.
<i>QuickDraw</i>	Si es verdadera, se utiliza la paleta global. El dibujo es más rápido, pero puede perderse calidad en ciertas resoluciones.

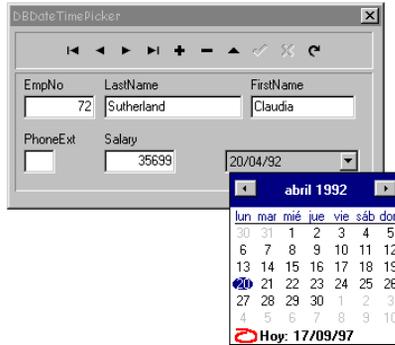
De forma similar a lo que ocurre con los campos de tipo memo, el componente *TDBImage* ofrece una propiedad *AutoDisplay* para acelerar la exploración de tablas que contienen imágenes. Cuando *AutoDisplay* está activo, cada vez que el conjunto de datos cambia su fila activa, cambia el contenido del control. Si es *False*, hay que pulsar INTRO sobre el control para visualizar el contenido del campo, o llamar al método *LoadPicture*.

Estos componentes permiten trabajar con el Portapapeles, utilizando las teclas estándar CTRL+C, CTRL+V y CTRL+X, o mediante los métodos *CopyToClipboard*, *CutToClipboard* y *PasteFromClipboard*.

La técnica del componente del pobre

C++ Builder nos ofrece un componente *TDateTimePicker*, que permite la edición y visualización de fechas como en un calendario. Este control se encuentra en la página *Win32* de la Paleta de Componentes. No existe ningún control similar para editar del mismo modo campos de bases de datos que contengan fechas ¿Qué hacemos en

estos casos, si no existe el componente en el mercado¹⁴ y no tenemos tiempo y paciencia para crear un nuevo componente? La solución consiste en utilizar los eventos del componente *TDataSource* para convertir a la pobre Cenicienta en una hermosa princesa, capaz de comunicarse con todo tipo de bases de datos:



Los eventos del componente *TDataSource* son los tres siguientes:

Evento	Se dispara...
<i>OnStateChange</i>	Cada vez que cambia el estado de la tabla asociada
<i>OnDataChange</i>	Cuando cambia el contenido del registro actual
<i>OnUpdateData</i>	Cuando hay que guardar los datos locales en el registro activo

Para ilustrar el uso del evento *OnStateChange*, cree la típica aplicación de prueba para tablas; ya sabe a qué me refiero:

- Una tabla (*TTable*) que podamos modificar sin remordimientos.
- Una fuente de datos (*TDataSource*) conectada a la tabla anterior.
- Una rejilla de datos (*TDBGrid*), para visualizar los datos. Asigne la fuente de datos a su propiedad *DataSource*.
- Una barra de navegación (*TDBNavigator*), para facilitarnos la navegación y edición. Modifique también su propiedad *DataSource*.

Vaya entonces a *DataSource1* y cree la siguiente respuesta al evento *OnStateChange*:

```

char* StateStr[] = {"Inactive", "Browse", "Edit", "Insert",
    "SetKey", "CalcFields", "Filter", "NewValue", "OldValue",
    "CurValue", "BlockRead", "InternalCalc"};

void __fastcall TForm1::DataSource1StateChange(TObject *Sender)
{
    static TForm *Ventana = NULL;
    static TListBox *Lista = NULL;

```

¹⁴ En realidad existen versiones de *TDBDateTimePicker* de todo tipo y color.

```

if (Ventana == NULL)
{
    Ventana = new TForm(this);
    Lista = new TListBox(Ventana);
    Lista->Align = alClient;
    Lista->Parent = Ventana;
    Ventana->Show();
}
Lista->Items->Add(StateStr[Table1->State]);
}
    
```

¿Y qué hay de la promesa de visualizar campos de fechas en un *TDateTimePicker*? Es fácil: cree otra ventana típica, con la tabla, la fuente de datos, una barra de navegación y unos cuantos *TDBEdit*. Utilice la tabla *employee.db* del alias *bcdemos*, por ejemplo. Esta tabla tiene un campo, *HireDate*, que almacena la fecha de contratación del empleado. Sustituya el cuadro de edición de este campo por un componente *TDateTimePicker*. Finalmente seleccione la fuente de datos y cree un manejador para el evento *OnDataChange*:

```

void __fastcall TForm1::DataSource1DataChange(TObject *Sender,
    TField *Field)
{
    DateTimePicker1->DateTime = Table1->FieldValues["HireDate"];
}
    
```

Antes he mencionado que *OnDataChange* se dispara cada vez que cambia el contenido del registro activo. Esto sucede, por ejemplo, cuando navegamos por la tabla, pero también cuando alguien (un control visual o un fragmento de código) cambia el valor de un campo. En el primer caso el parámetro *Field* del evento contiene *NULL*, mientras que en el segundo apunta al campo que ha sido modificado. Le propongo al lector que intente optimizar el método anterior haciendo uso de esta característica del evento.

Permitiendo las modificaciones

Con un poco de cuidado, podemos permitir también las modificaciones sobre el calendario. Necesitamos interceptar ahora el evento *OnUpdateData* de la fuente de datos:

```

void __fastcall TForm1::DataSource1UpdateData(TObject *Sender)
{
    Table1->FieldValues["HireDate"] = DateTimePicker1->DateTime;
}
    
```

Pero aún falta algo para que la maquinaria funcione. Y es que hemos insistido antes en que solamente se puede asignar un valor a un campo si la tabla está en estado de edición o inserción, mientras que aquí realizamos la asignación directamente. La respuesta es que necesitamos interceptar también el evento que anuncia los cambios

en el calendario; cuando se modifique la fecha queremos que el control ponga en modo de edición a la tabla de forma automática. Esto se realiza mediante el siguiente método:

```
void __fastcall TForm1::DateTimePicker1Change(TObject *Sender)
{
    DataSource1->Edit();
}
```

En vez de llamar al método *Edit* de la tabla o consulta asociada al control, he llamado al método del mismo nombre de la fuente de datos. Este método verifica primero si la propiedad *AutoEdit* del *data source* está activa, y si el conjunto de datos está en modo de exploración, antes de ponerlo en modo de edición.

De todos modos, tenemos un pequeño problema de activación recursiva. Cuando cambia el contenido del calendario, estamos poniendo a la tabla en modo de edición, lo cual dispara a su vez al evento *OnChange*, que relee el contenido del registro activo, y perdemos la modificación. Por otra parte, cada vez que cambia la fila activa, se dispara *OnChange*, que realiza una asignación a la fecha del calendario. Esta asignación provoca un cambio en el componente y dispara a *OnChange*, que pone entonces a la tabla en modo de edición. Esto quiere decir que tenemos que controlar que un evento no se dispare estando activo el otro. Para ello utilizaré una variable privada en el formulario, que actuará como si fuera un semáforo:

```
class TForm1 : public TForm
{
    // ...
private:
    bool FCambiando;
};

void __fastcall TForm1::DataSource1DataChange(TObject *Sender,
    TField *Field)
{
    if (! FCambiando)
    try
    {
        FCambiando = True;
        DateTimePicker1->DateTime = Table1->FieldValues["HireDate"];
    }
    __finally
    {
        FCambiando = False;
    }
}

void __fastcall TForm1::Calendar1Change(TObject *Sender)
{
    if (! FCambiando)
    try {
        FCambiando = True;
        DataSource1->Edit();
    }
}
```

```

    __finally {
        FCambiando = False;
    }
}

```

Finalmente, podemos interceptar también el evento *OnExit* del calendario, de modo que las modificaciones realizadas en el control se notifiquen a la tabla cuando abandonemos el calendario. Esto puede ser útil si tenemos más de un control asociado a un mismo campo, o si estamos visualizando a la vez la tabla en una rejilla de datos. La clave para esto la tiene el método *UpdateRecord* del conjunto de datos:

```

void __fastcall TForm1::DateTimePicker1Exit(TObject *Sender)
{
    Table1->UpdateRecord();
}

```

Blob, blob, blob...

Las siglas BLOB quieren decir en inglés, *Binary Large Objects*: Objetos Binarios Grandes. Con este nombre se conoce un conjunto de tipos de datos con las siguientes características:

- Longitud variable.
- Esta longitud puede ser bastante grande. En particular, en la arquitectura Intel puede superar la temible “barrera” de los 64 KB.
- En el caso general, el sistema de bases de datos no tiene por qué saber interpretar el contenido del campo.

La forma más fácil de trabajar con campos BLOB es leer y guardar el contenido del campo en ficheros. Se pueden utilizar para este propósito los métodos *LoadFromFile* y *SaveToFile*, de la clase *TBlobField*:

```

void __fastcall TBlobField::LoadFromFile(const AnsiString Fichero);
void __fastcall TBlobField::SaveToFile(const AnsiString Fichero);

```

El ejemplo típico de utilización de estos métodos es la creación de un formulario para la carga de gráficos en una tabla, si los gráficos residen en un fichero. Supongamos que se tiene una tabla, *imagenes*, con dos campos: *Descripcion*, de tipo cadena, y *Foto*, de tipo gráfico. En un formulario colocamos los siguientes componentes:

- tbImagenes* La tabla de imágenes.
- tbImagenesFoto* El campo correspondiente a la columna *Foto*.
- OpenDialog1* Cuadro de apertura de ficheros, con un filtro adecuado para cargar ficheros gráficos.

bnCargar Al pulsar este botón, se debe cargar una nueva imagen en el registro actual.

He mencionado solamente los componentes protagonistas; es conveniente añadir un *DBEdit* para visualizar el campo *Descripcion* y un *DBImage*, para la columna *Foto*; por supuesto, necesitaremos una fuente de datos. También será útil incluir una barra de navegación.

El código que se ejecuta en respuesta a la pulsación del botón de carga de imágenes debe ser:

```
void __fastcall TForm1::bnCargarClick(TObject *Sender)
{
    if (OpenDialog1.Execute())
    {
        if (!tbImágenes->State == dsEdit ||
            tbImágenes->State == dsInsert)
            tbImágenes->Edit();
        tbImágenesFoto->LoadFromFile(OpenDialog1->FileName);
    }
}
```

Observe que este método no depende de la presencia del control *TDBImage* para la visualización del campo. Me he adelantado en el uso del método *Edit*; sin esta llamada, la asignación de valores a campos provoca una excepción.

Otra posibilidad consiste en utilizar los métodos *LoadFromStream* y *SaveToStream* para transferir el contenido completo del campo hacia o desde un flujo de datos, en particular, un flujo de datos residente en memoria. Este tipo de datos, en C++ Builder, se representa mediante la clase *TMemoryStream*.

La clase *TBlobStream*

No hace falta, sin embargo, que el contenido del campo blob se lea completamente en memoria para poder trabajar con él. Para esto contamos con la clase *TBlobStream*, que nos permite acceder al contenido del campo como si estuviéramos trabajando con un fichero. Para crear un objeto de esta clase, hay que utilizar el siguiente constructor:

```
__fastcall TBlobStream::TBlobStream(TBlobField *Campo,
    TBlobStreamMode Modo);
```

El tipo *TBlobStreamMode*, por su parte, es un tipo enumerativo que permite los valores *bmRead*, *bmWrite* y *bmReadWrite*; el uso de cada uno de estos modos es evidente.

¿Quiere almacenar imágenes en formato JPEG en una base de datos? Desgraciadamente, el componente *TDBImage* no permite la visualización de este formato, que permite la compresión de imágenes. Pero no hay problemas, pues podemos almacenar estas imágenes en un campo BLOB sin interpretación, y encargarnos nosotros mismos de su captura y visualización. La captura de la imagen tendrá lugar mediante un procedimiento idéntico al que mostramos en la sección anterior para imágenes “normales”:

```
void __fastcall TForm1.bnCargarClick(TObject *Sender)
{
    if (OpenDialog1->Execute())
    {
        if (tbImagenes->State != dsEdit &&
            tbImagenes->State != dsEdit)
            tbImagenes->Edit();
        tbImagenesFoto->LoadFromFile(OpenDialog1->FileName);
    }
}
```

La única diferencia es que asumimos que el campo *Foto* de la tabla *tbImagenes* debe ser un campo BLOB sin interpretación, y que hemos configurado al componente *OpenDialog1* para que sólo nos permita abrir ficheros de extensión *jpeg* y *jpg*. Debemos incluir además el fichero de cabecera *jpeg.hpp* en el formulario.

Para visualizar las imágenes, traemos un componente *TImage* de la página *Additional* de la Paleta de Componentes, e interceptamos el evento *OnDataChange* de la fuente de datos asociada a *tbImagenes*:

```
void __fastcall TForm1::DataSource1DataChange(TObject *Sender,
    TField *Field)
{
    if (tbImagenesFoto->IsNull)
        Image1->Picture->Graphic = NULL;
    else
    {
        std::auto_ptr<TBlobStream> BS(
            new TBlobStream(tbImagenesFoto, bmRead));
        auto_ptr<TJPEGImage> Graphic(new TJPEGImage);
        Graphic->LoadFromStream(BS.get());
        Image1->Picture->Graphic = Graphic.get();
    }
}
```

La clase *TJPEGImage* está definida en la unidad *JPEG*, y descende del tipo *TGraphic*. El método anterior se limita a crear un objeto de esta clase y llenarlo con el contenido del campo utilizando un objeto *TBlobStream* como paso intermedio. Finalmente, el gráfico se asigna al control de imágenes.

Rejillas y barras de navegación

QUIZÁS LOS CONTROLES DE BASES DE DATOS MÁS POPULARES entre los programadores de Windows sean las rejillas y las barras de navegación. Las rejillas de datos nos permiten visualizar de una forma cómoda y general cualquier conjunto de datos. Muchas veces se utiliza una rejilla como punto de partida para realizar el mantenimiento de tablas. Desde la rejilla se pueden realizar búsquedas, modificaciones, inserciones y borrados. Las respuestas a consultas *ad hoc* realizadas por el usuario pueden también visualizarse en rejillas. Por otra parte, las barras de navegación son un útil auxiliar para la navegación sobre conjuntos de datos, estén representados sobre rejillas o sobre cualquier otro conjunto de controles. En este capítulo estudiaremos este par de componentes, sus propiedades y eventos básicos, y la forma de personalizarlos.

El uso y abuso de las rejillas

Sin embargo, es fácil utilizar incorrectamente las rejillas de datos. Pongamos por caso que una aplicación deba manejar una tabla de clientes de 1.000.000 de registros. El programador medio coloca una rejilla y ¡hala, a navegar! Si la aplicación está basada en tablas de Paradox o dBase no hay muchos problemas. Pero si tratamos con una base de datos SQL es casi seguro que se nos ahogue la red. Está claro que mostrar 25 filas en pantalla simultáneamente es más costoso que mostrar, por ejemplo, sólo un registro a la vez. Además, es peligroso dejar en manos de un usuario desaprensivo la posibilidad de moverse libremente a lo largo y ancho de un conjunto de datos. Si utilizamos el componente *TQuery*, en vez de *TTable*, para recuperar los datos y el usuario intenta ir al último registro del conjunto resultado, veremos cómo la red se pone literalmente al rojo vivo, mientras va trayendo al cliente cada uno de los 999.998 registros intermedios.

Quiero aclarar un par de puntos antes de continuar:

1. *No siempre es posible limitar el tamaño de un conjunto resultado.* Existen técnicas que estudiaremos más adelante, como los filtros y rangos, que permiten reducir el número de filas de una tabla. Y está claro que utilizando una cláusula **where** en

una consulta podemos lograr el mismo efecto. Pero esta reducción de tamaño no siempre es suficiente. Por ejemplo, ¿cómo limitamos la vista de clientes? ¿Por la inicial del apellido? Vale, hay 26 letras, con lo que obtendremos una media de 40.000 registros. ¿Por ciudades? Tampoco nos sirve. ¿Limitamos el resultado a los 1000 primeros registros? Recuerde que el parámetro *MAX ROWS* nos permite hacerlo, pero esta limitación es demasiado arbitraria para tener sentido.

2. *Puede ser imprescindible mostrar varias filas a la vez.* Es frecuente oír el siguiente argumento: ¿para qué utilizar una rejilla sobre 1.000.000 de registros, si sólo vamos a poder ver una pequeña ventana de 25 filas a la vez? ¿Es posible sacar algo en claro de una rejilla, que no podamos averiguar navegando registro por registro? Yo creo que sí. Muchas veces olvidamos que la navegación con rejillas cobra especial importancia cuando ordenamos la tabla subyacente por alguna de sus columnas. En tal caso, el análisis del contexto en que se encuentra determinado registro puede aportarnos bastante información. Nos puede ayudar a detectar errores ortográficos en un nombre, que cierto empleado se encuentra en una banda salarial especial, etc.

Un error que comete la mayoría de los programadores consiste en sobrecargar una rejilla con más columnas de lo debido. Esto es un fallo de diseño, pues una rejilla en la que hay que realizar desplazamientos horizontales para ver todas las columnas es poco menos que inútil. Busque, por ejemplo, la libreta de direcciones de su aplicación de correo electrónico. Lo más probable es que los nombres y apellidos de los destinatarios de correo aparezcan en una lista o rejilla, y que el resto de sus datos puedan leerse de uno en uno, en otros controles de edición. Bien, ese es el modelo que le propongo de uso de rejillas. Casi siempre, las columnas que muestro en una rejilla corresponden a la clave primaria o a una clave alternativa. Es posible también que incluya alguna otra columna de la cual quiera obtener información contextual: esto implica con toda seguridad que la rejilla estará ordenada de acuerdo al valor de esa columna adicional. El resto de los campos los situó en controles *data-aware* orientados a campos: cuadros de edición, combos, imágenes, etc.

El funcionamiento básico de una rejilla de datos

Para que una rejilla de datos “funcione”, basta con asignarle una fuente de datos a su propiedad *DataSource*. Es todo. Quizás por causa de la sencillez de uso de estos componentes, hay muchos detalles del uso y programación de rejillas de datos que el desarrollador normalmente pasa por alto, o descuida explicar en su documentación para usuarios. Uno de estos descuidos es asumir que el usuario conoce todos los detalles de la interfaz de teclado y ratón de este control. Y es que esta interfaz es rica y compleja.

Las teclas de movimiento son las de uso más evidente. Las flechas nos permiten movernos una fila o un carácter a la vez, podemos utilizar el avance y retroceso de

página; las tabulaciones nos llevan de columna en columna, y es posible usar la tabulación inversa.

La tecla *INS* pone la tabla asociada a la rejilla en modo de inserción. Aparentemente, se crea un nuevo registro con los valores por omisión, y el usuario debe llenar el mismo. Para grabar el nuevo registro tenemos que movernos a otra fila. Por supuesto, si tenemos una barra de navegación asociada a la tabla, el botón *Post* produce el mismo efecto sin necesidad de cambiar la fila activa. Un poco más adelante estudiaremos las barras de navegación.

Pulsando *F2*, el usuario pone a la rejilla en modo de edición; *C++ Builder* crea automáticamente un cuadro de edición del tamaño de la celda activa para poder modificar el contenido de ésta. Esta acción también se logra automáticamente cuando el usuario comienza a teclear sobre una celda. La *edición automática* se controla desde la propiedad *AutoEdit* de la fuente de datos (*data source*) a la cual se conecta la rejilla. Para grabar los cambios realizados hacemos lo mismo que con las inserciones: pulsamos el botón *Post* de una barra de navegación asociada o nos cambiamos de fila.

Otra combinación útil es *CTRL+SUPR*, mediante la cual se puede borrar el registro activo en la rejilla. Cuando hacemos esto, se nos pide una confirmación. Es posible suprimir este mensaje, que es lanzado por la rejilla, y pedir una confirmación personalizada para cada tabla interceptando el evento *BeforeDelete* de la propia tabla. Esto se explicará en el capítulo 28, sobre eventos de transición de estados.

La rejilla de datos tiene una columna fija, en su extremo izquierdo, que no se mueve de lugar aún cuando nos desplazamos a columnas que se encuentran fuera del área de visualización. En esta columna, la fila activa aparece marcada, y la marca depende del estado en que se encuentre la tabla base. En el estado *dsBrowse*, la marca es una punta de flecha; cuando estamos en modo de edición, una viga I (*i-beam*), la forma del cursor del ratón cuando se sitúa sobre un cuadro de edición; en modo de inserción, la marca es un asterisco. Como veremos, esta columna puede ocultarse manipulando las opciones de la rejilla.

Por otra parte, con el ratón podemos cambiar en tiempo de ejecución la disposición de las columnas de una rejilla, manipulando la barra de títulos. Por ejemplo, arrastrando una cabecera de columna se cambia el orden de las columnas; arrastrando la división entre columnas, se cambia el tamaño de las mismas. A partir de *C++ Builder 3* pueden incluso utilizarse las cabeceras de columnas como botones. Naturalmente, la acción realizada en respuesta a esta acción debe ser especificada por el usuario interceptando un evento.

Opciones de rejillas

Muchas de las características visuales y funcionales de las rejillas pueden cambiarse mediante la propiedad *Options*. Aunque las rejillas de datos, *TDbGrid* y las rejillas *TDrawGrid* y *TStringGrid* están relacionadas entre sí, las opciones de estas clases son diferentes. He aquí las opciones de las rejillas de datos y sus valores por omisión:

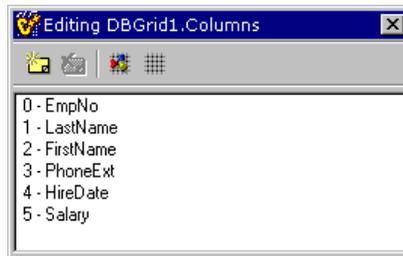
Opción	PO	Significado
<i>dgEditing</i>	Sí	Permite la edición de datos sobre la rejilla
<i>dgAlwaysShowEditor</i>	No	Activa siempre el editor de celdas
<i>dgTitles</i>	Sí	Muestra los títulos de las columnas
<i>dgIndicator</i>	Sí	La primera columna muestra el estado de la tabla
<i>dgColumnResize</i>	Sí	Cambiar el tamaño y posición de las columnas
<i>dgColLines</i>	Sí	Dibuja líneas entre las columnas
<i>dgRowLines</i>	Sí	Dibuja líneas entre las filas
<i>dgTabs</i>	Sí	Utilizar tabulaciones para moverse entre columnas
<i>dgRowSelect</i>	No	Seleccionar filas completas, en vez de celdas
<i>dgAlwaysShowSelection</i>	No	Dejar siempre visible la selección
<i>dgConfirmDelete</i>	Sí	Permite confirmar los borrados
<i>dgCancelOnExit</i>	Sí	Cancela inserciones vacías al perder el foco
<i>dgMultiSelect</i>	No	Permite seleccionar varias filas a la vez.

Muchas veces, es conveniente cambiar las opciones de una rejilla en coordinación con otras opciones o propiedades. Por ejemplo, cuando queremos que una rejilla se utilice sólo en modo de lectura, además de cambiar la propiedad *ReadOnly* es aconsejable eliminar la opción *dgEditing*. De este modo, cuando el usuario seleccione una celda, no se creará el editor sobre la celda y no se llevará a impresión de que la rejilla iba a permitir la modificación. Como ejemplo adicional, cuando preparamos una rejilla para seleccionar múltiples filas con la opción *dgMultiSelect*, es bueno activar también la opción *dgRowSelect*, para que la barra de selección se dibuje sobre toda la fila, en vez de sobre celdas individuales.

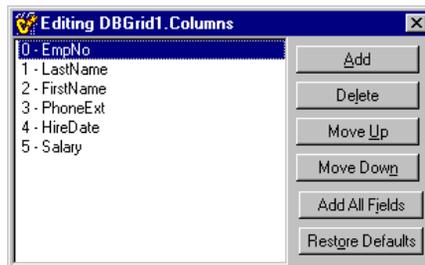
Columnas a la medida

La configuración de una rejilla de datos va más allá de las posibilidades de la propiedad *Options*. Por ejemplo, es necesario indicar el orden inicial de las columnas, los títulos, la alineación de los textos dentro de las columnas... En la VCL 1, anterior a C++ Builder, las tareas mencionadas se llevaban a cabo modificando propiedades de los componentes de campos de la tabla asociada a la rejilla. Por ejemplo, si queríamos cambiar el título de una columna, debíamos modificar la propiedad *DisplayLabel* del campo correspondiente. La alineación de la columna se extraía de la propiedad *Alignment* del campo. Y si queríamos ocultar una columna, debíamos utilizar la propiedad *Visible* del componente de campo.

Esto ocasionaba bastantes problemas; el problema más grave era lo limitado de las posibilidades de configuración según este estilo. Por ejemplo, aunque un campo se alineara a la derecha, el título de su columna se alineaba siempre a la izquierda. A partir de la versión 2 de la VCL las cosas hubieran podido agravarse, por causa de la aparición de los módulos de datos, que permiten utilizar el mismo componente no visual de acceso con diferentes modos de visualización. Al colocar una tabla determinada en un módulo de datos y configurar las propiedades visuales de un componente de campo en dicho módulo, cualquier rejilla que se conectara a la tabla mostraría la misma apariencia. Para poder separar la parte visual de los métodos de acceso, se hizo indispensable la posibilidad de configurar directamente las rejillas de datos.



La propiedad *Columns* permite modificar el diseño de las columnas de una rejilla de datos. El tipo de esta propiedad es *TDBGridColumns*, y es una colección de objetos de tipo *TColumn*. Para editar esta propiedad podemos hacer un doble clic en el valor de la propiedad en el Inspector de Objetos, o realizar el doble clic directamente sobre la propia rejilla. El aspecto del editor de propiedades es diferente en C++ Builder 3:



Las propiedades que nos interesan de las columnas son:

Propiedad	Significado
<i>Alignment</i>	Alineación de la columna
<i>ButtonStyle</i>	Permite desplegar una lista desde una celda, o mostrar un botón de edición.
<i>Color</i>	Color de fondo de la columna.
<i>DropDownRows</i>	Número de filas desplegables.
<i>FieldName</i>	El nombre del campo que se visualiza.

Propiedad	Significado
<i>Font</i>	El tipo de letra de la columna.
<i>PickList</i>	Lista opcional de valores a desplegar.
<i>ReadOnly</i>	Desactiva la edición en la columna.
<i>Width</i>	El ancho de la columna, en píxeles.
<i>Title.Alignment</i>	Alineación del título de la columna.
<i>Title.Caption</i>	Texto de la cabecera de columna.
<i>Title.Color</i>	Color de fondo del título de columna.
<i>Title.Font</i>	Tipo de letra del título de la columna.

C++ Builder 4 añade a esta lista la propiedad *Expanded*, que se aplica a las columnas que representan campos de objetos de Oracle. Si *Expanded* es *True*, la columna se divide en subcolumnas, para representar los atributos del objeto, y la fila de títulos de la rejilla duplica su ancho, para mostrar tanto el nombre de la columna principal como los nombres de las dependientes. Esta propiedad puede modificarse tanto en tiempo de diseño como en ejecución.

Si el programador no especifica columnas en tiempo de diseño, éstas se crean en tiempo de ejecución y se llenan a partir de los valores extraídos de los campos de la tabla; observe que algunas propiedades de las columnas se corresponden a propiedades de los componentes de campo. Si existen columnas definidas en tiempo de diseño, son éstas las que se utilizan para el formato de la rejilla.

En la mayoría de las situaciones, las columnas se configuran en tiempo de diseño, pero es también posible modificar propiedades de columnas en tiempo de ejecución. El siguiente método muestra como se pueden mostrar de forma automática en color azul las columnas de una rejilla que pertenezcan a los campos que forman parte del índice activo.

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    for (int i = 0; i < DBGrid1->Columns->Count; i++)
    {
        TColumn *C = DBGrid1->Columns->Items[i];
        if (C->Field->IsIndexField)
            C->Font->Color = clBlue;
    }
}
```

En este otro ejemplo tenemos una rejilla con dos columnas, que ocupa todo el espacio interior de un formulario. Queremos que la segunda columna de la rejilla ocupe todo el área que deja libre la primera columna, aún cuando cambie el tamaño de la ventana. Se puede utilizar la siguiente instrucción:

```
void __fastcall TForm1::FormResize(TObject *Sender)
{
    DBGrid1->Columns->Items[1]->Width = DBGrid1->ClientWidth -
```

```

        DBGrid1->Columns->Items[0]->Width - IndicatorWidth - 2;
    }

```

El valor que se resta en la fórmula, *IndicatorWidth*, corresponde a una variable global declarada en la unidad *DBGrids*, y corresponde al ancho de la columna de indicadores. He restado 2 píxeles para tener en cuenta las líneas de separación. Si la rejilla cambia sus opciones de visualización, cambiará el valor a restar, por supuesto.

Para saber qué columna está activa en una rejilla, utilizamos la propiedad *SelectedIndex*, que nos dice su posición. *SelectedField* nos da acceso al componente de campo asociado a la columna activa. Por otra parte, si lo que queremos es la lista de campos de una rejilla, podemos utilizar la propiedad vectorial *Fields* y la propiedad entera *FieldCount*. Un objeto de tipo *TColumn* tiene también, en tiempo de ejecución, una propiedad *Field* para trabajar directamente con el campo asociado.

Guardar y restaurar los anchos de columnas

Para los usuarios de nuestras aplicaciones puede ser conveniente poder mantener el formato de una rejilla de una sesión a otra, especialmente los anchos de las columnas. Voy a mostrar una forma sencilla de lograrlo, suponiendo que cada columna de la rejilla pueda identificarse de forma única por el nombre de su campo asociado. El ejemplo utiliza ficheros de configuración, pero puede adaptarse fácilmente para hacer uso del registro de Windows.

Supongamos que el formulario *Form1* tiene una rejilla *DBGrid1* en su interior. Entonces necesitamos la siguiente respuesta al evento *OnCreate* del formulario para restaurar los anchos de columnas de la sesión anterior:

```

const AnsiString SSlaveApp = "Software\\MiEmpresa\\MiAplicacion\\";

void __fastcall TForm1::FormCreate(TObject *Sender)
{
    std::auto_ptr<TRegIniFile> ini(new TRegIniFile(
        SSlaveApp + "Rejillas\\" + Name + "." + DBGrid1->Name));
    for (int i = 0; i < DBGrid1->Columns->Count; i++)
    {
        TColumn c = DBGrid1->Columns->Items[i];
        c->Width = ini->ReadInteger("Width", c->FieldName, c->Width);
    }
}

```

Estamos almacenando los datos de la rejilla *DBGrid1* en la siguiente clave del registro de Windows:

```
[HKEY_CURRENT_USER\MiEmpresa\MiAplicacion\Rejillas\Form1.DBGrid1]
```

El tercer parámetro de *ReadInteger* es el valor que se debe devolver si no se encuentra la clave dentro de la sección. Este valor se utiliza la primera vez que se ejecuta el programa, cuando aún no existe el fichero de configuraciones. Este fichero se debe actualizar cada vez que se termina la sesión, durante el evento *OnClose* del formulario:

```
void __fastcall TForm1::FormClose(TObject *Sender,
    TCloseAction &Action)
{
    std::auto_ptr<TRegIniFile> ini(new TRegIniFile(
        SSlaveApp + "Rejillas\\" + Name + "." + DBGrid1->Name));
    for (int i = 0; i < DBGrid1->Columns->Count; i++)
    {
        TColumn *C = DBGrid1->Columns->Items[i];
        ini->WriteInteger("Width", C->FieldName, C->Width);
    }
}
```

Sobre la base de estos procedimientos simples, el lector puede incorporar mejoras, como la lectura de la configuración por secciones completas, y el almacenamiento de más información, como el orden de las columnas.

Listas desplegables y botones de edición

Las rejillas de datos permiten que una columna despliegue una lista de valores para que el usuario seleccione uno de ellos. Esto puede suceder en dos contextos diferentes:

- Cuando el campo visualizado en una columna es un campo de búsqueda (*lookup field*).
- Cuando el programador especifica una lista de valores en la propiedad *PickList* de una columna.

En el primer caso, el usuario puede elegir un valor de una lista que se extrae de otra tabla. El estilo de interacción es similar al que utilizan los controles *TDBLookupComboBox*, que estudiaremos más adelante; no se permite, en contraste, la búsqueda incremental mediante teclado. En el segundo caso, la lista que se despliega contiene exactamente los valores tecleados por el programador en la propiedad *PickList* de la columna. Esto es útil en situaciones en las que los valores más frecuentes de una columna se reducen a un conjunto pequeño de posibilidades: formas de pagos, fórmulas de tratamiento (Señor, Señora, Señorita), y ese tipo de cosas. En cualquiera de estos casos, la altura de la lista desplegable se determina por la propiedad *DropDownRows*: el número máximo de filas a desplegar.

PartNo	VendorNo	Vendor	Description	Cost	ListPrice
900	3820	Techniques	Dive kayak	\$1.356,75	\$3.999,95
912	3820	J.W. Luscher Mfg	Underwater Diver Vehicle	\$504,00	\$1.680,00
1313	3511	Scuba Professional Divers' Supply Sho	Regulator System	\$117,50	\$250,00
1314	5641	Techniques	Second Stage Regulator	\$124,10	\$365,00
1316	3511	Perry Scuba	Regulator System	\$119,35	\$341,00
1320	3511	Beauchat, Inc. Amor Agua	Second Stage Regulator	\$73,53	\$171,00
1328	3511	Scuba Professionals	Regulator System	\$154,80	\$430,00
1330	3511	Scuba Professionals	Alternate Inflation Regulator	\$85,80	\$260,00

La propiedad *ButtonStyle* determina si se utiliza el mecanismo de lista desplegable o no. Si vale *bsAuto*, la lista se despliega si se cumple alguna de las condiciones anteriores. Si la propiedad vale *bsNone*, no se despliega nunca la lista. Esto puede ser útil en ocasiones: suponga que hemos definido, en la tabla que contiene las líneas de detalles de un pedido, el precio del artículo que se vende como un campo de búsqueda, que partiendo del código almacenado en la línea de detalles, extrae el precio de venta de la tabla de artículos. En este ejemplo, no nos interesa que se despliegue la lista con todos los precios existentes, y debemos hacer que *ButtonStyle* valga *bsNone* para esa columna.

Por otra parte, si asignamos el valor *bsEllipsis* a la propiedad *ButtonStyle* de alguna columna, cuando ponemos alguna celda de la misma en modo de edición aparece en el extremo derecho de su interior un pequeño botón con tres puntos suspensivos. Lo único que hace este botón es lanzar el evento *OnEditButtonClick* cuando es pulsado:

```
void __fastcall TForm1::DBGrid1EditButtonClick(TObject *Sender)
```

La columna en la cual se produjo la pulsación es la columna activa de la rejilla, que se puede identificar por su posición, *SelectedIndex*, o por el campo asociado, *SelectedField*. Este botón también puede activarse pulsando CTRL+INTRO.

En C++ Builder 4 los puntos suspensivos aparecen también con los campos *TReferenceField* y *TDataSetField*, de Oracle 8. Cuando se pulsa el botón, aparece una rejilla emergente con los valores anidados dentro del campo. El nuevo método *ShowPopupEditor* permite invocar al editor desde el programa.

Números verdes y números rojos

La propiedad *Columns* nos permite especificar un color para cada columna por separado. ¿Qué sucede si deseamos, por el contrario, colores diferentes por fila, o incluso por celdas? Y puestos a pedir, ¿se pueden dibujar gráficos en una rejilla? Claro que sí: para eso existe el evento *OnDrawColumnCell*.

Comencemos por algo sencillo: en la tabla de inventario *parts.db* queremos mostrar en color rojo aquellos artículos para los cuales hay más pedidos que existencias; la tabla en cuestión tiene sendas columnas, *OnOrder* y *OnHand*, para almacenar estas cantidades. Así que creamos un manejador para *OnDrawColumnCell*, y C++ Builder nos presenta el siguiente esqueleto de método:

```
void __fastcall TForm1::DBGrid1DrawColumnCell(TObject *Sender,
    const TRect Rect, int DataCol, TColumn *TColumn,
    TGridDrawState State)
{
}
```

Rect es el área de la celda a dibujar, *DataCol* es el índice de la columna a la cual pertenece y *Column* es el objeto correspondiente; por su parte, *State* indica si la celda está seleccionada, enfocada y si es una de las celdas de cabecera. En ninguna parte se nos dice la fila que se va a dibujar, pero la tabla asociada a la rejilla tendrá activo el registro correspondiente durante la respuesta al evento. Así que podemos empezar por cambiar las condiciones de dibujo: si el valor del campo *OnOrder* iguala o supera al valor del campo *OnHand* en la fila activa de la tabla, cambiamos el color del tipo de letras seleccionado en el lienzo de la rejilla a rojo. Después, para dibujar el texto ... un momento, ¿no estamos complicando un poco las cosas?

La clave para evitar este dolor de cabeza es el método *DefaultDrawColumnCell*, perteneciente a las rejillas de datos. Este método realiza el dibujo por omisión de las celdas, y puede ser llamado en el interior de la respuesta a *OnDrawColumnCell*. Los parámetros de este método son exactamente los mismos que se suministran con el evento; de este modo, ni siquiera hay que consultar la ayuda en línea para llamar al método. Si el manejador del evento se limita a invocar a este método, el dibujo de la rejilla sigue siendo idéntico al original. Podemos entonces limitarnos a cambiar las condiciones iniciales de dibujo, realizando asignaciones a las propiedades del lienzo de la rejilla, antes de llamar a esta rutina. He aquí el resultado:

```
void __fastcall TForm1::DBGrid1DrawColumnCell(TObject *Sender,
    const TRect Rect, int DataCol, TColumn *TColumn,
    TGridDrawState State)
{
    TDBGrid *grid = static_cast<TDBGrid*>(Sender);
    if (tbParts->FieldValues["OnOrder"] >=
        tbParts->FieldValues["OnHand"])
        grid->Canvas->Font->Color = clRed;
    grid->DefaultDrawColumnCell(Rect, DataCol, Column, State);
}
```

Por supuesto, también podemos dibujar el contenido de una celda sin necesidad de recurrir al dibujo por omisión. Si estamos visualizando la tabla de empleados en una rejilla, podemos añadir desde el Editor de Columnas una nueva columna, con el botón *New*, dejando vacía la propiedad *FieldName*. Esta columna se dibujará en blanco. Añadimos también al formulario un par de componentes de imágenes, *TImage*, con

los nombres *CaraAlegre* y *CaraTriste*, y mapas de bits que hagan juego; estos componentes deben tener la propiedad *Visible* a *False*. Finalmente, interceptamos el evento de dibujo de celdas de columnas:

```
void __fastcall TForm1::DBGrid1DrawColumnCell(TObject *Sender,
const TRect Rect, int DataCol, TColumn *TColumn,
TGridDrawState State)
{
    TDBGrid *grid = static_cast<TDBGrid*>(Sender);
    if (Column->FieldName != "")
        grid->DefaultDrawColumnCell(Rect, DataCol, Column, State);
    else if (tbEmpleados->FieldValues["Salary"] >= 45000)
        grid->Canvas->StretchDraw(Rect, CaraAlegre->Picture->Graphic);
    else
        grid->Canvas->StretchDraw(Rect, CaraTriste->Picture->Graphic);
}
```

EmpNo	LastName	FirstName	PhoneExt	HireDate	Salary
2	Nelson	Roberto	250	28/12/88	40000
4	Young	Bruce	233	28/12/88	55500
5	Lambert	Kim	22	6/02/89	25000
8	Johnson	Leslie	410	5/04/89	25050
9	Forest	Phil	229	17/04/89	25090
11	Weston	K. J.	34	17/01/90	33292.9375
12	Lee	Terri	256	1/05/90	45332
14	Hall	Stewart	227	4/06/90	34482.625
15	Young	Katherine	231	14/06/90	24400
20	Papadopoulos	Chris	887	1/01/90	25050
24	Fisher	Pete	888	12/09/90	23040
28	Bennet	Ann	5	1/02/91	34482.8

Otro ejemplo, que quizás sea más práctico, es mostrar el valor de un campo lógico dentro de una rejilla como una casilla de verificación. Supongamos que la tabla *Table1* contiene un campo de nombre *Activo*, de tipo lógico. Para dibujar la columna correspondiente al campo en la rejilla, utilizamos el siguiente método en respuesta al evento *OnDrawColumnCell*:

```
void __fastcall TForm1::DBGrid1DrawColumnCell(TObject *Sender,
const TRect Rect, int DataCol, TColumn *TColumn,
TGridDrawState State)
{
    if (CompareText(Column->FieldName, "ACTIVO") == 0)
    {
        UINT check = 0;
        if (Table1->FieldValues["ACTIVO"])
            check = DFCS_CHECKED;
        DBGrid1->Canvas->FillRect(Rect);
        DrawFrameControl(DBGrid1->Canvas->Handle, (RECT*) &Rect,
            DFC_BUTTON, DFCS_BUTTONCHECK | check);
    }
    else
        DBGrid1->DefaultDrawColumnCell(Rect, DataCol, Column, State);
}
```

DrawFrameControl es una función del API de Windows que nos permite dibujar muchos de los controles nativos. Recuerde que mientras más complicado sea el dibujo, más tardará en redibujarse la rejilla.

Más eventos de rejillas

En C++ Builder 3 se añadieron un par de nuevos eventos a las rejillas de datos. Estos son *OnCellClick*, que se dispara cuando el usuario pulsa el ratón sobre una celda de datos, y *OnTitleClick*, cuando la pulsación ocurre en alguno de los títulos de columnas. Aunque estos eventos pueden detectarse teóricamente directamente con los eventos de ratón, *OnCellClick* y *OnTitleClick* nos facilitan las cosas al pasar, como parámetro del evento, el puntero al objeto de columna donde se produjo la pulsación.

Para demostrar el uso de estos eventos, coloque en un formulario vacío una tabla con las siguientes propiedades:

Propiedad	Valor
<i>DatabaseName</i>	<i>IBLOCAL</i>
<i>TableName</i>	<i>EMPLOYEE</i>
<i>Active</i>	<i>True</i>

Es importante para este ejemplo que la tabla pertenezca a una base de datos SQL; es por eso que utilizamos los ejemplos de InterBase. Coloque en el formulario, además, un *TDataSource* y una rejilla de datos, debidamente conectados.

Luego, cree la siguiente respuesta al evento *OnTitleClick* de la rejilla:

```
void __fastcall TForm1::DBGrid1TitleClick(TColumn *Column)
{
    try
    {
        if (Column->Field->FieldKind == fkLookup)
            Table1->IndexFieldNames = Column->Field->KeyFields;
        else
            Table1->IndexFieldNames = Column->FieldName;
    }
    catch(Exception&)
    {
    }
}
```

La propiedad *IndexFieldNames* de las tablas se utiliza para indicar por qué campo, o combinación de campos, queremos que la tabla esté ordenada. Para una tabla SQL este campo puede ser arbitrario, cosa que no ocurre para las tablas locales; en el capítulo sobre índices trataremos este asunto. Nuestra aplicación, por lo tanto, permite

cambiar el criterio de ordenación de la tabla que se muestra con sólo pulsar con el ratón sobre el título de la columna por la cual se quiere ordenar.

La barra de desplazamiento de la rejilla

La otra gran diferencia entre las rejillas de datos de C++ Builder 1 y las de versiones posteriores consiste en el comportamiento de la barra de desplazamiento vertical que tienen asociadas. En las versiones 1 y 2 de la VCL, para desesperación de muchos programadores habituados a trabajar con bases de datos locales, la barra solamente asume tres posiciones: al principio, en el centro y al final. ¿Por qué? La culpa la tienen las tablas SQL: para saber cuántas filas tiene una tabla residente en un servidor remoto necesitamos cargar todas las filas en el ordenador cliente. ¿Y todo esto sólo para que el cursor de la barra de desplazamiento aparezca en una posición proporcional? No merece la pena, y pagan justos por pecadores, pues también se utiliza el mismo mecanismo para las tablas de Paradox y dBase.

Afortunadamente, C++ Builder 3 corrigió esta situación para las tablas locales aunque, por supuesto, las cosas siguen funcionando igual para las bases de datos SQL.

Rejillas de selección múltiple

Como hemos visto, si activamos la opción *dgMultiSelect* de una rejilla, podemos seleccionar varias filas simultáneamente sobre la misma. La selección múltiple se logra de dos formas diferentes:

- Extendiendo la selección mediante las flechas hacia arriba y hacia abajo, manteniendo pulsada la tecla de mayúsculas.
- Pulsando con el ratón sobre una fila, mientras se sostiene la tecla CTRL.

Si pulsamos CTRL+SUPR mientras la rejilla tiene el foco del teclado, eliminaremos todas las filas seleccionadas. Cualquier otra operación que deseemos deberá ser programada.

La clave para programar con una rejilla de selección múltiple es la propiedad *SelectedRows*. Esta propiedad es del tipo *TBookmarkList*, una lista de marcas de posición, cuyos principales métodos y propiedades son los siguientes:

Propiedades/Métodos	Propósito
<i>Count</i>	Cantidad de filas seleccionadas.
<i>Items</i> [<i>int index</i>]	Lista de posiciones seleccionadas.
<i>CurrentRow.Selected</i>	¿Está seleccionada la fila activa?

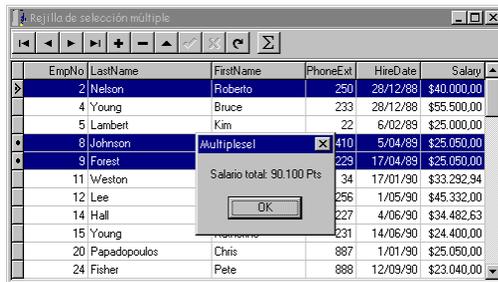
Propiedades/Métodos	Propósito
<code>void __fastcall Clear();</code>	Eliminar la selección.
<code>void __fastcall Delete();</code>	Borrar las filas seleccionadas.
<code>void __fastcall Refresh();</code>	Actualizar la selección, eliminando filas borradas.

El siguiente método muestra cómo sumar los salarios de los empleados seleccionados en una rejilla con selección múltiple:

```

void __fastcall TForm1::bnSumarClick(TObject *Sender)
{
    Currency Total = 0;
    tbEmpleados->DisableControls();
    AnsiString BM = tbEmpleados->Bookmark;
    try
    {
        for (int I = 0; I < DBGrid1->SelectedRows->Count; I++)
        {
            tbEmpleados->Bookmark = DBGrid1->SelectedRows->Items[I];
            Total += tbEmpleados->FieldValues["Salary"];
        }
    }
    finally
    {
        tbEmpleados->Bookmark = BM;
        tbEmpleados->EnableControls();
    }
    ShowMessage(Format("Salario total: %m", ARRAYOFCONST((Total))));
}
    
```

La técnica básica consiste en controlar el recorrido desde un bucle **for**, e ir activando sucesivamente cada fila seleccionada en la lista de marcas. Para que la tabla se mueva su cursor al registro marcado en la rejilla, solamente necesitamos asignar la marca a la propiedad *Bookmark* de la tabla.



Barras de navegación

Al igual que sucede con las rejillas de datos, la principal propiedad de las barras de navegación es *DataSource*, la fuente de datos controlada por la barra. Cuando esta propiedad está asignada, cada uno de los botones de la barra realiza una acción sobre

el conjunto de datos asociado a la fuente de datos: navegación (*First, Prior, Next, Last*), inserción (*Insert*), eliminación (*Delete*), modificación (*Edit*), confirmación o cancelación (*Post, Cancel*) y actualización de los datos en pantalla (*Refresh*).



Un hecho curioso: muchos programadores me preguntan en los cursos que imparto si se pueden modificar las imágenes de los botones de la barra de navegación. Claro que se puede, respondo, pero siempre me quedo intrigado, pues no logro imaginar un conjunto de iconos más “expresivo” o “adecuado”. ¿Acaso flechas *art nouveau* verdes sobre fondo rojo? De todos modos, para el que le interese este asunto, las imágenes de los botones están definidas en el fichero *dbctrls.res*, que se encuentra en el subdirectorio *lib* de C++ Builder. Se puede cargar este fichero con cualquier editor gráfico de recursos, Image Editor incluido, y perpetrar el correspondiente atentado contra la estética.

También he encontrado programadores que sustituyen completamente la barra de navegación por botones de aceleración. Esta técnica es correcta, y es fácil implementar tanto las respuestas a las pulsaciones de los botones como mantener el estado de activación de los mismos; hemos visto cómo se hace esto último al estudiar los eventos del componente *TDataSource* en el capítulo anterior. Sin embargo, existe una razón poderosa para utilizar siempre que sea posible la barra de navegación de C++ Builder, al menos sus cuatro primeros botones, y no tiene que ver con el hecho de que ya esté programada. ¿Se ha fijado lo que sucede cuando se deja pulsado uno de los botones de navegación durante cierto tiempo? El comando asociado se repite entonces periódicamente. Implementar este comportamiento desde cero ya es bastante más complejo, y no merece la pena.

Había una vez un usuario torpe, muy torpe...

...tan torpe que no acertaba nunca en el botón de la barra de navegación que debía llevarlo a la última fila de la tabla. No señor: este personaje siempre “acertaba” en el botón siguiente, el que inserta registros. Por supuesto, sus tablas abundaban en filas vacías ... y la culpa, según él, era del programador. Que nadie piense que lo estoy inventando, es un caso real.

Para estas situaciones tenemos la propiedad *VisibleButtons* de la barra de navegación. Esta propiedad es la clásica propiedad cuyo valor es un conjunto. Podemos, por ejemplo, esconder todos los botones de actualización de una barra, dejando solamente los cuatro primeros botones. Esconder botones de una barra provoca un desagradable efecto secundario: disminuye el número de botones pero el ancho total del componente permanece igual, lo que conduce a que el ancho individual de cada

botón aumente. Claro, podemos corregir la situación posteriormente reduciendo el ancho general de la barra.

A propósito de botones, las barras de navegación tienen una propiedad *Flat*, para que el borde tridimensional de los botones esté oculto hasta que el ratón pase por encima de uno de ellos. La moda ejerce su dictadura también en las pantallas de nuestros ordenadores.

Ayudas para navegar

Aunque la barra de navegación tiene una propiedad *Hint* como casi todos los controles, esta propiedad no es utilizada por C++ Builder. Las indicaciones por omisión que muestran los botones de una barra de navegación estaban definidas, en C++ Builder 1, en el fichero de recursos *dbconsts.res*, y era posible editar este fichero para cambiar los valores en él almacenados. En las versiones 3 y 4, se definen en la unidad *dbconsts.pas*, utilizando las cadenas de recursos (**resourcestring**) de Delphi.

Para personalizar las indicaciones de ayuda, es necesario utilizar la propiedad *Hints*, en plural, que permite especificar una indicación por separado para cada botón. *Hints* es de tipo *TStrings*, una lista de cadenas. La primera cadena corresponde al primer botón, la segunda cadena, que se edita en la segunda línea del editor de propiedades, corresponde al segundo botón, y así sucesivamente. Esta correspondencia se mantiene incluso cuando hay botones no visibles. Por supuesto, las ayudas asociadas a los botones ausentes no tendrán efecto alguno.

El comportamiento de la barra de navegación

Una vez modificada la apariencia de la barra, podemos también modificar parcialmente el comportamiento de la misma. Para esto contamos con el evento *OnClick* de este componente:

```
void __fastcall TForm1::DBNavigator1Click(TObject *Sender,
    TNavigateButton Button)
{
}
```

Podemos ver que, a diferencia de la mayoría de los componentes, este evento *OnClick* tiene un parámetro adicional que nos indica qué botón de la barra ha sido pulsado. Este evento se dispara *después* de que se haya efectuado la acción asociada al botón; si se ha pulsado el botón de editar, el conjunto de datos asociado ya ha sido puesto en modo de edición. Esto se ajusta al concepto básico de tratamiento de eventos: un evento es un contrato sin obligaciones. No importa si no realizamos acción alguna en

respuesta a un evento en particular, pues el mundo seguirá girando sin nuestra cooperación.

Mostraré ahora una aplicación de este evento. Según mi gusto personal, evito en lo posible que el usuario realice altas y modificaciones directamente sobre una rejilla. Prefiero, en cambio, que estas modificaciones se efectúen sobre un cuadro de diálogo modal, con los típicos botones de aceptar y cancelar. Este diálogo de edición debe poderse ejecutar desde la ventana en la que se efectúa la visualización mediante la rejilla de datos. Si nos ceñimos a este estilo de interacción, no nos vale el comportamiento normal de las barras de navegación. Supongamos que *Form2* es el cuadro de diálogo que contiene los controles necesarios para la edición de los datos visualizados en el formulario *Form1*. Podemos entonces definir la siguiente respuesta al evento *OnClick* de la barra de navegación existente en *Form1*:

```
void __fastcall TForm1::DBNavigator1Click(TObject *Sender,
    TNavigateButton Button)
{
    if (Button == nbEdit || Button == nbInsert)
        // La tabla está ya en modo de edición o inserción
        Form2->ShowModal();
}
```

De este modo, al pulsar el botón de edición o el de inserción, se pone a la tabla base en el estado correspondiente y se activa el diálogo de edición. Hemos supuesto que este diálogo tiene ya programadas acciones asociadas a los botones para grabar o cancelar los cambios cuando se cierra. Podemos incluso aprovechar los métodos de clase mostrados en el capítulo de técnicas de gestión de ventanas para crear dinámicamente este cuadro de diálogo:

```
void __fastcall TForm1::DBNavigator1Click(TObject *Sender,
    TNavigateButton Button)
{
    if (Button == nbEdit || Button == nbInsert)
        // La tabla está ya en modo de edición o inserción
        TForm2::Mostrar(__classid(TForm2),
            Button == nbEdit ? 0 : 1); // Creación dinámica
}
```

Un método útil es el siguiente:

```
void __fastcall TDBNavigator::BtnClick(TNavigateBtn Index);
```

Este método simula la pulsación del botón indicado de la barra de navegación. Supongamos que, en el ejemplo anterior, quisiéramos que una doble pulsación del ratón sobre la rejilla de datos activase el diálogo de edición para modificar los datos de la fila actual. En vez de programar a partir de cero esta respuesta, lo más sensato es aprovechar el comportamiento definido para la barra de navegación. Interceptamos de la siguiente forma el evento *OnDbClick* de la rejilla de datos:

```
void __fastcall TForm1::DBGrid1DbClick(TObject *Sender)
{
    DBNavigator1->BtnClick(nbEdit);
}
```

Observe que la llamada al método *BtnClick* va a disparar también el evento asociado a *OnClick* de la barra de navegación.

El evento *BeforeAction* es disparado cuando se pulsa un botón, pero antes de que se produzca la acción asociada al mismo. El prototipo del evento es similar al de *OnClick*. A veces yo utilizo este evento para cambiar la acción asociada al botón de inserción. Este botón *inserta* visualmente una fila entre la fila actual y la anterior, pero en muchos casos es más interesante que la fila vaya al final de la rejilla, directamente. Es decir, quiero que el botón ejecute el método *Append*, no *Insert*. Bueno, ésta es una forma de lograrlo:

```
void __fastcall TForm1::DBNavigator1BeforeAction(TObject *Sender)
{
    static_cast<TDBNavigator*>(Sender)->DataSource->
        DataSet->Append();
    SysUtils::Abort();
}
```

Rejillas de controles

Un *TDBGrid* de C++ Builder no puede editar, al menos directamente, un campo lógico como si fuera una casilla de verificación. Es igualmente cierto que, mediante los eventos *OnDrawColumnCell*, *OnCellClick* y una gran dosis de buena voluntad, podemos simular este comportamiento. Pero también podemos utilizar el componente *TDBCtrlGrid*, que nos permite, mediante la copia de controles de datos individuales, mostrar varios registros a la vez en pantalla.

En principio, un *TDBCtrlGrid* aparece dividido en paneles, y uno de estos paneles acepta otros controles en su interior. En tiempo de ejecución, los otros paneles, que aparecen inactivos durante el diseño, cobran vida y repiten en su interior controles similares a los colocados en el panel de diseño. En cada uno de estos paneles, por supuesto, se muestran los datos correspondientes a un registro diferente. Las propiedades que tienen que ver con la apariencia y disposición de los paneles son:

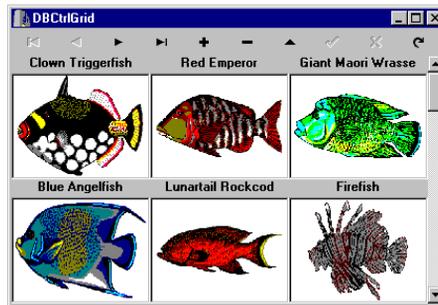
Propiedad	Significado
<i>Orientation</i>	Orientación del desplazamiento de paneles.
<i>ColCount</i>	Número de columnas.
<i>RowCount</i>	Número de filas.
<i>PanelBorder</i>	¿Tiene borde el panel? ¹⁵

¹⁵ ¿Y sueñan los androides con ovejas eléctricas?

Propiedad	Significado
<i>PanelWidth</i>	Ancho de cada panel, en píxeles.
<i>PanelHeight</i>	Altura de cada panel, en píxeles.

Las rejillas de controles tienen la propiedad *DataSource*, que indica la fuente de datos a la cual están conectados los diferentes controles de su interior. Cuando un control de datos se coloca en este tipo de rejilla debe “renunciar” a tener una fuente de datos diferente a la del *TDBCtrlGrid*. De este modo pueden incluirse cuadros de edición, combos, casillas de verificación, memos, imágenes o cualquier control que nos dicte nuestra imaginación. La excepción son los *TDBRadioGroup* y los *TDBRichEdit*.

En C++ Builder 1, sin embargo, no se podían colocar directamente componentes *DBMemo* y *DBImage* en una rejilla de controles. La causa inmediata era que estos controles carecían de la opción *csReplicatable* dentro de su conjunto de opciones *ControlStyle*. La causa verdadera era que el BDE no permitía el uso de caché para campos BLOB. No obstante, se pueden crear componentes derivados que añadan la opción de replicación durante la creación.



La implementación de componentes duplicables en la VCL es algo complicada y no está completamente documentada. Aunque el tema sale fuera del alcance de este libro, mencionaré que estos componentes deben responder al mensaje interno *CM_GETDATALINK*, y tener en cuenta el valor de la propiedad *ControlState* para dibujar el control, pues si el estado *csPaintCopy* está activo, hay que leer directamente el valor a visualizar desde el campo.

UNO DE LOS ASPECTOS BÁSICOS de la implementación de todo sistema de bases de datos es la definición y uso de índices. Los índices están presentes en casi cualquier área de la creación y explotación de una base de datos:

- Mediante los índices se pueden realizar búsquedas rápidas.
- Las búsquedas rápidas se aprovechan en la implementación de las restricciones de integridad referencial.
- Si los índices se implementan mediante árboles balanceados (*b-trees*) o alguna técnica equivalente, que es lo más frecuente, nos sirven también para realizar diferentes ordenaciones lógicas sobre los registros de las tablas.
- Sobre un índice que permite una ordenación se pueden definir eficientemente rangos, que son restricciones acerca de los registros visibles en un momento determinado.
- Gracias a las propiedades anteriores, los índices se utilizan para optimizar las operaciones SQL tales como las selecciones por un valor y los encuentros entre tablas.

En el tercer punto de la lista anterior, he mencionado indirectamente la posibilidad de índices que no estén implementados mediante árboles balanceados. Esto es perfectamente posible, siendo el caso más típico los índices basados en técnicas de *hash*, o desmenuzamiento de la clave. Estos índices, sin embargo, no permiten la ordenación de los datos básicos. Por esto, casi todos los índices de los sistemas de bases de datos más populares están basados en árboles balanceados y permiten tanto la búsqueda rápida como la ordenación.

Con qué índices podemos contar

El primer paso para trabajar con índices en C++ Builder es conocer qué índices tenemos asociados a una tabla determinada. Si lo único que nos importa es la lista de nombres de índices, podemos utilizar el método *GetIndexNames*, aplicable a la clase *TTable*. Su prototipo es:

```
void __fastcall TTable::GetIndexNames(TStrings *Lista);
```

Este método añade a la lista pasada como parámetro los nombres de los índices que están definidos para esta tabla. Es importante advertir que la acción realizada por *GetIndexNames* es añadir nuevas cadenas a una lista existente, por lo que es necesario limpiar primero la lista, con el método *Clear*, si queremos obtener un resultado satisfactorio. Para que *GetIndexNames* funcione no hace falta que la tabla esté abierta; un objeto de tipo *TTable* necesita solamente tener asignado valores a las propiedades *DatabaseName* y *TableName* para poder preguntar por los nombres de sus índices. Otra particularidad de este método es que no devuelve en la lista el índice primario de las tablas Paradox.

Ahora bien, si necesitamos más información sobre los índices, es en la propiedad *IndexDefs* de la tabla donde debemos buscarla. Esta propiedad, que devuelve un objeto de tipo *TIndexDefs*, es similar en cierto sentido a la propiedad *FieldDefs*, que ya hemos analizado. Podemos utilizar *IndexDefs* incluso con la tabla cerrada, pero para eso tenemos primeramente que aplicar el método *Update* al objeto retornado por esta propiedad. La clase *TIndexDefs* es en esencia una lista de definiciones de índices; cada definición de índice es, a su vez, un objeto de clase *TIndexDef*, en singular. Para acceder a las definiciones individuales se utilizan estas dos subpropiedades de *TIndexDefs*:

```
__property int Count;
__property TIndexDef *Items[int I];
```

Por su parte, estas son las propiedades de un objeto de clase *TIndexDef*:

Propiedad	Significado
<i>Name</i>	Es el nombre del índice; si éste es el índice primario de una tabla Paradox, la propiedad contiene una cadena vacía.
<i>Fields</i>	Lista de campos de la clave, separados por puntos y comas. Si es un índice de expresiones de dBase, la propiedad contiene una cadena vacía.
<i>Expression</i>	Expresión que define un índice de expresiones de dBase.
<i>Options</i>	Conjunto de opciones del índice.

La propiedad *Options* es un conjunto al cual pueden pertenecer los siguientes valores:

Opción	Significado
<i>ixPrimary</i>	El índice es el primario de una tabla Paradox.
<i>ixUnique</i>	No se permiten claves duplicadas en el índice.
<i>ixDescending</i>	El criterio de ordenación es descendente.
<i>ixExpression</i>	El índice es un índice de expresiones de dBase.
<i>ixCaseInsensitive</i>	Se ignora la diferencia entre mayúsculas y minúsculas. No aplicable en tablas dBase.

Si no existiera el método *GetIndexNames* se podría simular con el siguiente procedimiento:

```
void __fastcall LeerNombresDeIndices(TTable *Table,
    TStrings *Lista)
{
    Table->IndexDefs->Update();
    Lista->Clear(); // Una mejora al algoritmo
    for (int i = 0; i < Table->IndexDefs->Count; i++)
    {
        TIndexDef *id = Table->IndexDefs->Items[i];
        if (id->Name != "")
            Lista->Add(id->Name);
    }
}
```

Pero si lo que deseamos es obtener una lista de las definiciones de índices, similar a la lista desplegable de la propiedad *IndexFieldNames* que estudiaremos en breve, debemos utilizar un procedimiento parecido a este otro:

```
void __fastcall LeerDefiniciones(TTable *Tabla, TStrings* Lista)
{
    TIndexDefs *ID = Tabla->IndexDefs;
    ID->Update();
    Lista->Clear();
    for (int I = 0; I < ID->Count; I++)
        if (ID->Items[I]->Options.Contains(ixExpression))
            Lista->Add(ID->Items[I]->Expression);
        else
            Lista->Add(ID->Items[I]->Fields);
}
```

Especificando el índice activo

Ahora que podemos conocer qué índices tiene una tabla, nos interesa empezar a trabajar con los mismos. Una de las operaciones básicas con índices en C++ Builder es especificar el *índice activo*. El índice activo determina el orden lógico de los registros de la tabla. Es necesario tener un índice activo para poder realizar las siguientes operaciones:

- Búsquedas rápidas con *FindKey*, *FindNearest* y otros métodos relacionados.
- Activación de rangos.
- Uso de componentes *TDBLookupComboBox* y *TDBLookupListBox* con la tabla asignada indirectamente en su propiedad *ListSource*.
- Tablas en relación *master/detail*. La tabla de detalles debe indicar un índice activo. Esto lo veremos más adelante.

Aunque normalmente en una aplicación de entrada de datos el criterio de ordenación de una tabla es el mismo para toda la aplicación, el cambio de índice activo es una operación posible y completamente dinámica. A pesar de que los manuales aconsejan cerrar la tabla antes de cambiar de índice, esta operación puede realizarse con la tabla abierta sin ningún tipo de problemas, en cualquier momento y con un costo despreciable.

Existen dos formas de especificar un índice para ordenar una tabla, y ambas son mutuamente excluyentes. Se puede indicar el nombre de un índice existente en la propiedad *IndexName*, o se pueden especificar los campos que forman el índice en la propiedad *IndexFieldNames*. Si utilizamos *IndexName* debemos tener en cuenta que los índices primarios de Paradox no aparecen en la lista desplegable del editor de la propiedad. Si queremos activar este índice debemos asignar una cadena vacía a la propiedad: este es, por otra parte, su valor por omisión. La propiedad *IndexFieldNames*, en cambio, es más fácil de utilizar, pues en la lista desplegable de su editor de propiedad aparecen los campos que forman la clave de cada índice; generalmente, esto es más informativo que el simple nombre asignado al índice.



En el siguiente ejemplo muestro cómo cambiar en tiempo de ejecución el orden de una tabla utilizando un combo con la lista de criterios posibles de ordenación. Para el ejemplo se necesita, por supuesto, una tabla (*Table1*) y un cuadro de combinación (*ComboBox1*) cuya propiedad *Style* sea igual a *csDropDownList*. Es necesario definir manejadores para el evento *OnCreate* de la tabla y para el evento *OnChange* del combo:

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    // Llamar a la función del epígrafe anterior
    LeerDefiniciones(Table1, ComboBox1->Items);
    // Seleccionar el primer elemento del combo
    ComboBox1->ItemIndex = 0;
    // Simular el cambio de selección en el combo
    ComboBox1Change(ComboBox1);
}
```

```

void __fastcall TForm1::ComboBox1Change(TObject *Sender)
{
    Table1->IndexName =
        Table1->IndexDefs[ComboBox1->ItemIndex]->Name;
}

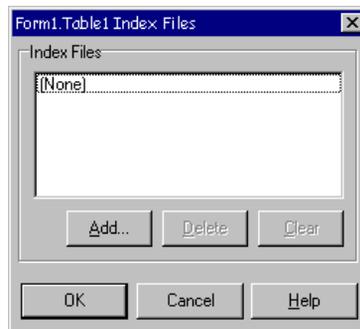
```

Observe que, en vez de asignar directamente a *IndexFieldNames* los campos almacenados en el texto del combo, se busca el nombre del índice correspondiente en *IndexDefs*. El motivo son los índices de expresiones de dBase; recuerde que la función *LeerDefiniciones* fue programada para mostrar la expresión que define al índice en estos casos.

Aunque algunos sistemas SQL permiten definir índices descendentes, como InterBase, la activación de los mismos mediante la propiedad *IndexName* no funciona correctamente, pues se utiliza siempre el orden ascendente. La única forma de lograr que los datos aparezcan ordenados descendentemente es utilizar una consulta *TQuery*. Sin embargo, las consultas presentan determinados inconvenientes, sobre los que trataremos más adelante en el capítulo 25, sobre comunicación cliente/servidor.

Indices en dBase

En dBase es posible, aunque no aconsejable, la existencia de varios ficheros *mdx* asociados a una tabla. Estos índices secundarios no se actualizan automáticamente. La razón fundamental de esta restricción consiste en que al abrir el fichero principal *dbf* no existe forma de saber, sin intervención del programador, si tiene algún fichero *mdx* secundario asociado.



Si necesita de todos modos este tipo de índices debe utilizar la propiedad *IndexFiles* de las tablas. Esta propiedad, de tipo *TStrings*, almacena la lista de nombres de ficheros *mdx* y *ndx* (¡sí, los de dBase III!) que deben abrirse junto a la tabla para su mantenimiento. Normalmente, esta propiedad debe asignarse en tiempo de diseño, pero es

posible añadir y eliminar dinámicamente elementos en esta propiedad. El mismo resultado se obtiene mediante los métodos *OpenIndexFile* y *CloseIndexFile*, de la clase *TTable*. Cualquier componente de un índice secundario abierto puede seleccionarse, en la propiedad *IndexName*, para utilizarlo como índice activo de la tabla.

Especificando un orden en tablas SQL

Si estamos trabajando con una tabla perteneciente a una base de datos orientada a registros (una forma elegante de decir Paradox ó dBase), solamente podemos utilizar los criterios de ordenación inducidos por los índices existentes. Si quiero ordenar una tabla de empleados por sus salarios y no existe el índice correspondiente mis deseos quedarán insatisfechos. Sin embargo, si la base de datos es una base de datos SQL, puedo ordenar por la columna o combinación de columnas que se me antoje; es responsabilidad del servidor la ordenación según el método más eficiente. Y para esto se utiliza la propiedad *IndexFieldNames*. En esta propiedad se puede asignar el nombre de una columna de la tabla, o una lista de columnas separadas por puntos y comas. Por supuesto, las columnas deben pertenecer a tipos de datos que permitan la comparación. El orden utilizado es el ascendente.

Se puede realizar demostrar esta técnica si visualizamos una tabla de InterBase o cualquier otro sistema SQL en una rejilla. En el formulario creamos un menú emergente, de tipo *TPopupMenu*, con una opción *Ordenar*. En el evento *OnPopup* programamos lo siguiente:

```
void __fastcall TForm1::PopupMenuPopup(TObject *Sender)
{
    AnsiString S = ';' + Table1.IndexFieldNames + ';';
    Ordenar1->Checked =
        S.Pos('; ' + DBGrid1.SelectedField.FieldName + ';') != 0;
}
```

De esta forma el comando de menú aparece inicialmente marcado si el nombre del campo correspondiente a la columna seleccionada se encuentra ya en el criterio de ordenación. Para añadir o eliminar el campo del criterio de ordenación, creamos un manejador para el evento *OnClick* del comando:

```
void __fastcall TForm1::Ordenar1Click(TObject *Sender)
{
    AnsiString Criterio = ";" + Table1->IndexFieldNames + ";";
    AnsiString Campo = ";" + DBGrid1->SelectedField->FieldName + ";";
    int Posicion = Criterio.Pos(Campo);
    if (Posicion != 0)
    {
        Criterio.Delete(Posicion, Campo.Length() - 1);
        Table1->IndexFieldNames = Criterio.SubString(2,
            Criterio.Length() - 2);
    }
}
```

```

else if (Criterio == ";")
    Table1->IndexFieldNames = DBGrid1->SelectedField->FieldName;
else
    Table1->IndexFieldNames = Table1->IndexFieldNames
        + ";" + DBGrid1->SelectedField->FieldName;
}

```

Le propongo al lector que mejore esta técnica añadiendo al menú emergente la posición del campo en la lista de columnas, si es que está ordenado.

Búsqueda basada en índices

En C++ Builder es muy fácil realizar una búsqueda rápida sobre el índice activo. Al estar los índices principales de todos los formatos de bases de datos soportados por el BDE implementados mediante árboles balanceados o técnicas equivalentes, es posible realizar dos tipos de búsqueda: la búsqueda exacta de un valor y la búsqueda inexacta en la cual, si no encontramos el valor deseado, nos posicionamos en la fila correspondiente al valor más cercano en la secuencia ascendente o descendente de claves del índice. Estas dos operaciones, a su vez, se pueden realizar en C++ Builder utilizando directamente ciertos métodos simples o descomponiendo estos métodos en secuencias de llamadas más elementales; por supuesto, esta segunda técnica es más complicada. Comenzaremos por la forma más sencilla y directa, para luego explicar el por qué de la existencia de la segunda.

Para realizar una búsqueda exacta sobre un índice activo se utiliza el método *FindKey*. Su prototipo es:

```

bool __fastcall TTable::FindKey(const TVarRec *KeyValues,
    const int KeyValues_Size);

```

El parámetro *Valores* corresponde a la clave que deseamos buscar. Como la clave puede ser compuesta y estar formada por campos de distintos tipos, se utiliza un vector de valores de tipo arbitrario para contenerla. Ya hemos visto el mecanismo de la VCL para listas de parámetros en el capítulo que trata sobre los tipos de datos de C++ Builder.

Si *FindKey* encuentra la clave especificada, la tabla a la cual se le aplica el método cambia su fila activa a la fila que contiene el valor. En ese caso, *FindKey* devuelve *True*. Si la clave no se encuentra, devuelve *False* y no se cambia la fila activa.

Por el contrario, el método *FindNearest* no devuelve ningún valor, porque siempre encuentra una fila:

```

void __fastcall TTable::FindNearest(const TVarRec *KeyValues,
    const int KeyValues_Size);

```

El propósito de *FindNearest* es encontrar el registro cuyo valor de la clave coincide con el valor pasado como parámetro. Sin embargo, si la clave no se encuentra en la tabla, el cursor se mueve a la primera fila cuyo valor es superior a la clave especificada. De este modo, una inserción en ese punto dejaría espacio para un registro cuya clave fuera la suministrada al método.

La aplicación más evidente del método *FindNearest* es la búsqueda incremental. Supongamos que tenemos una tabla ordenada por una clave alfanumérica y que la estamos visualizando en una rejilla. Ahora colocamos en el formulario un cuadro de edición, de tipo *TEdit*, y hacemos lo siguiente en respuesta al evento *OnChange*:

```
void __fastcall TForm1::Edit1Change(TObject *Sender)
{
    Table1->FindNearest(ARRAYOFCONST((Edit1->Text)));
}
```

Se puede evitar el uso del cuadro de edición si la rejilla está en modo de sólo lectura:

Propiedad	Valor
<i>ReadOnly</i>	<i>True</i>
<i>Options</i>	Eliminar <i>dgEditing</i>

Necesitamos también declarar una variable de cadena para almacenar la clave de búsqueda actual; ésta se declara en la sección **private** de la declaración de la clase del formulario:

```
private:
    AnsiString Clave; // Se inicializa a cadena vacía
```

Después hay que interceptar el evento *OnKeyPress* de la rejilla:

```
void __fastcall TForm1::DBGrid1KeyPress(TObject *Sender, char &Key)
{
    if (Key >= ' ')
        AppendStr(Clave, Key);
    else if (Key == 8)
        Clave.Delete(Clave.Length(), 1);
    else
        return;
    Key = 0;
    Table1->FindNearest(ARRAYOFCONST((Clave)));
}
```

El inconveniente principal de esta técnica es que no sabemos en un momento determinado qué clave hemos tecleado exactamente, pero se puede utilizar un texto estático, *TLabel*, para paliar este problema.

Código	Apellidos	Nombre	Extensión	Contrato	Salario
24	Baldwin	Janet	2	21/03/91	\$23.900,00
105	Bender	Olive H.	255	8/10/92	\$36.793,00
26	Bennet	Ann	5	1/02/91	\$34.482,80
83	Bishop	Diana	290	1/06/92	\$45.000,00
109	Brown	Kelly	202	4/02/93	\$27.000,00
71	Butenk	Jennifer M.	269	15/04/92	\$45.932,00
107	Cook	Kevin	694	1/02/93	\$35.900,00
29	De Souza	Roger	268	18/02/91	\$25.500,00
121	Fenail	Roberto	1	12/07/93	\$40.500,00
24	Fisher	Pete	888	12/09/90	\$23.040,00
9	Forest	Phil	229	17/04/89	\$25.060,00
134	Glan	Jacques		23/08/93	\$24.895,00
138	Green	T.J.	218	1/11/93	\$36.000,00

Implementación de referencias mediante *FindKey*

Como sabemos, los campos de referencia tienen una implementación directa y sencilla en C++ Builder. Sin embargo, puede que en ciertas ocasiones nos interese prescindir de esta técnica e implementar de forma manual la traducción de un código a una descripción.

Volvemos al ejemplo de los pedidos de la base de datos *bcdemos*. La tabla *items* representa las cantidades vendidas de un producto en determinado pedido. En cada fila de esta tabla se almacena únicamente el código del producto, *PartNo*, por lo cual para conocer el precio por unidad del mismo hay que realizar una búsqueda en la tabla de artículos, *parts*. El problema consiste en que necesitamos la descripción del producto y su precio. Si creamos dos campos de referencia, C++ Builder realizará dos búsquedas sobre la misma tabla, cuando en realidad solamente necesitamos una búsqueda.

Por lo tanto, crearemos dos campos calculados, llamémosle *Descripcion* y *PrecioUnitario*, sobre la tabla *items*. El primero será de tipo *String* y el segundo de tipo *Currency*. Para calcular sus valores interceptamos el evento *OnCalcFields* del siguiente modo:

```
void __fastcall TForm1.tbItemsCalcFields(TDataSet *Sender)
{
    if (tbParts->FindKey(ARRAYOFCONST((tbItemsPartNo->Value)))
    {
        tbItemsDescripcion->Value = tbPartsDescription->Value;
        tbItemsPrecioUnitario->Value = tbPartsListPrice->Value;
    }
}
```

He supuesto que están creados ya los campos para cada una de las dos tablas utilizadas.

Este código tiene un par de detalles importantes para asimilar. En primer lugar, la tabla *parts* tiene que tener al índice primario, definido sobre el código de producto, como índice activo. En segundo lugar, este algoritmo cambia la posición de la fila

activa de la tabla de artículos. Es por ello que no debe utilizarse esta misma tabla para visualizar los artículos, pues el usuario verá como el cursor de la misma se mueve desenfrenadamente de vez en cuando. En realidad, el algoritmo utilizado por la VCL para los campos de referencia utiliza la función *Lookup*, de más fácil manejo, que veremos un poco más adelante.

Búsquedas utilizando *SetKey*

Ahora vamos a descomponer la acción de los métodos *FindKey* y *FindNearest* en llamadas a métodos de más bajo nivel. El método fundamental de esta técnica es *SetKey*:

```
void __fastcall TTable::SetKey();
```

El objetivo de este método es muy simple: colocar a la tabla en el estado especial *dsSetKey*. En este estado se permiten asignaciones a los campos de la tabla; estas asignaciones se interpretan como asignaciones a una especie de *buffer* de búsqueda. Por supuesto, estas asignaciones deben realizarse sobre los campos que componen la clave del índice activo. Una vez que los valores deseados se encuentran en el *buffer* de búsqueda, podemos utilizar uno de los métodos siguientes para localizar un registro:

```
bool __fastcall TTable::GotoKey();
void __fastcall TTable::GotoNearest();
```

La correspondencia de estos métodos con *FindKey* y *FindNearest* es evidente. *GotoKey* intentará localizar una fila que corresponda exactamente a la clave especificada, y *GotoNearest* localizará siempre la más cercana. Si deseamos salir del estado *dsSetKey* sin realizar búsqueda alguna, podemos utilizar el método *Cancel* para regresar al estado normal: el estado *dsBrowse*.

Experimentando con *SetKey*

Para comprender mejor el uso de *SetKey* le propongo un pequeño experimento, no muy útil como técnica de interfaz, pero que aclara el sentido de este método. Para el experimento necesitamos un formulario con una rejilla de exploración. Supongamos además que la tabla visualizada es la tabla de empleados del alias *bcdemos*, la tabla *employee.db*. Los objetos protagonistas son:

Objeto	Propiedad	Valor
<i>Form1</i>		(El formulario principal)
<i>Table1</i>		(La tabla de empleados)
	<i>DatabaseName</i>	<i>bcdemos</i>
	<i>TableName</i>	<i>employee.db</i>

	<i>IndexName</i>	<i>ByName</i>	
<i>DataSource1</i>	<i>DataSet</i>	<i>Table1</i>	(Fuente de datos para <i>Table1</i>)
<i>DBGrid1</i>	<i>DataSource</i>	<i>DataSource1</i>	(Rejilla de exploración)
<i>Button1</i>			(Para invocar el diálogo de búsqueda)

El índice *ByName* utilizado en la tabla *employee* está definido sobre las columnas *LastName*, el apellido, y *FirstName*, el nombre del empleado. El botón que hemos colocado permitirá invocar a un cuadro de búsqueda. En este segundo formulario colocamos los siguientes objetos:

Objeto	Propiedad	Valor	
<i>Form2</i>			(El formulario de búsqueda)
<i>DBEdit1</i>	<i>DataSource</i>	<i>Form1->DataSource1</i>	(Nombre del empleado)
	<i>DataField</i>	<i>FirstName</i>	
<i>DBEdit2</i>	<i>DataSource</i>	<i>Form1->DataSource1</i>	(Apellido del empleado)
	<i>DataField</i>	<i>LastName</i>	
<i>Button1</i>	<i>Kind</i>	<i>bkOk</i>	(Botón para aceptar)
<i>Button2</i>	<i>Kind</i>	<i>bkCancel</i>	(Botón para cancelar)

Observe que los cuadros de edición están trabajando directamente con la misma tabla que la rejilla del primer formulario; ésta es la parte fundamental del experimento.

El único código que necesitamos en este ejemplo se produce en respuesta a la pulsación del botón del primer formulario:

```
void __fastcall TForm1.Button1Click(TObject *Sender)
{
    Table1->SetKey();
    if (Form2->ShowModal() == mrOk)
        Table1->GotoNearest();
    else
        Table1->Cancel();
}
```

Cuando se pulsa el botón la tabla se pone en el estado *dsSetKey* y se invoca al cuadro de búsqueda que hemos creado. Si mueve este cuadro y observa la rejilla verá como desaparecen las filas de la misma temporalmente. Ahora, cualquier cosa que tecleemos en los cuadros de edición del diálogo será considerada como una asignación a un campo de la tabla, aunque en realidad se trata de una asignación a un *buffer* de

búsqueda: mientras tecleamos podemos ver también el efecto sobre el contenido de la rejilla. Cuando por fin cerramos el diálogo cancelamos la búsqueda o la disparamos, en dependencia del botón utilizado para terminar el diálogo, y entonces la tabla regresa a la normalidad. Repito, no es una técnica para incluir directamente en un programa, pero es interesante para comprender el mecanismo de búsquedas mediante el índice activo.

¿Por qué existe *SetKey*?

Además de cualquier motivo filosófico o espiritual que puedan alegar ciertos libros de C++ Builder que circulan por ahí, *SetKey* existe porque existen los índices de expresiones de dBase; de no ser por esta razón, *SetKey* pudiera quedar como un método interno de la implementación de la VCL. En este tipo de índices, *FindKey* y *FindNearest* se niegan rotundamente a trabajar: el primer paso de estos métodos es “repartir” los valores de lista pasada como parámetro entre los campos que forman el índice. Pero en un índice de expresiones no es sencillo determinar cuáles son los campos que forman el índice; en realidad es algo que no se intenta en absoluto.

Por ejemplo, tomemos una tabla que tenga los campos *Nombre* y *Apellidos*, y cuyo índice activo sea un índice basado en la expresión *Nombre + Apellidos*. Hay dos campos involucrados en el índice, y un programador inocente puede verse tentado a programar algo así:

```
// No funciona
Table1->FindKey(ARRAYOFCONST(("Howard", "Lovecraft")));
```

¿Cuál es el valor que debe tomarse como nombre y cuál debe tomarse como apellido? C++ Builder no puede saberlo. Y tanto *FindKey* como *FindNearest* están programados para lanzar una excepción si el índice activo es un índice de expresiones. La técnica correcta para realizar una búsqueda sobre este tipo de índices es la siguiente:

```
Table1->SetKey();
Table1->FieldValues["Nombre"] = "Howard";
Table1->FieldValues["Apellidos"] = "Lovecraft";
Table1->GotoNearest();
```

Observe que el último método de la secuencia es *GotoNearest*, en vez de *GotoKey*. La causa es la misma: para determinar si encontramos la fila buscada tendríamos que ser capaces de descomponer la expresión del índice en campos, y esto puede no ser posible, en el caso general.

Existe una variante de *SetKey*, el método *EditKey*, que es útil solamente cuando el índice activo, o el criterio de ordenación, incluye varias columnas. *EditKey* coloca la

tabla en el estado *dsSetKey*, pero no borra las asignaciones anteriores en el *buffer* de búsqueda. De este modo, después de efectuar la búsqueda del ejemplo anterior podemos ejecutar las siguientes instrucciones para buscar a un tal Howard Duncan:

```
Table1->EditKey();
Table1->FieldValues["Apellidos"] = "Duncan";
Table1->GotoNearest();
```

Rangos: desde el Alfa a la Omega

Un rango en C++ Builder es una restricción de las filas visibles de una tabla, mediante la cual se muestran solamente las filas en las cuales los valores de ciertas columnas se encuentran entre dos valores dados. La implementación de este recurso se basa en los índices, por lo cual para poder establecer un rango sobre una o varias columnas debe existir un índice sobre las mismas y estar activo. No podemos definir rangos simultáneamente sobre dos índices diferentes; si tenemos un índice sobre nombres de empleados y otro sobre salarios, no podemos utilizar rangos para pedir las personas cuyos nombres comienzan con la letra A y que ganen entre tres y cinco millones de pesetas anuales. Eso sí, podemos establecer una de estas restricciones y luego utilizar alguna otra técnica, como los filtros que veremos más adelante, para volver a limitar el resultado.

El índice sobre el cual se define un rango puede ser un índice compuesto, definido sobre varias columnas. Sin embargo, la semántica de la aplicación de rangos sobre índices compuestos es bastante confusa e induce fácilmente a errores. Lo digo por experiencia personal, pues en la edición anterior de este libro di una explicación equivocada de cómo funciona esta operación. El ejemplo concreto que utilicé fue éste: tenemos una tabla *Table1* que está ordenada por un índice compuesto por el nombre y el apellido. ¿Qué hace la siguiente instrucción?

```
Table1->SetRange(ARRAYOFCONST(("M", "F")),
ARRAYOFCONST(("Mzzzz", "Fzzzz")));
```

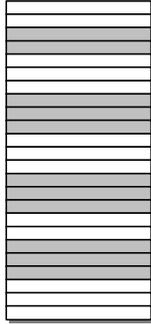
Al parecer, debe limitar el conjunto de registros activos a aquellas personas cuyo nombre comienza con “M” y su apellido con “F”. ¡No! Ahora aparecen, además de *María Filiberta*, casi todas las personas cuyo nombre comienza con “M” ... sin importar el apellido, al parecer. El problema es que uno espera los registros que satisfacen la condición:

```
'M' <= Nombre and Nombre <= 'Mzzzz' and
'F' <= Apellidos and Apellidos <= 'Fzzzz'
```

Pero lo que C++ Builder genera es lo siguiente:

```
(('M' = Nombre and 'F' <= Apellidos) or 'M' < Nombre) and
((Nombre = 'Mzzzz' and Apellidos <= 'Fzzzz') or Nombre <= 'Mzzzz')
```

Es decir, el criterio sobre el apellido solamente se aplica *en los extremos* del rango definido por el criterio establecido sobre el nombre. La siguiente imagen muestra la situación:



Esto es lo que esperamos...



.. y esto es lo que obtenemos...

Los filtros, que serán estudiados en el siguiente capítulo, nos permiten restringir el conjunto de registros activos utilizando criterios independientes para cada campo.

Lo mismo que sucede con las búsquedas sobre índices activos, sucede con los rangos: existen métodos de alto nivel y de bajo nivel, y la razón es la misma. La forma más fácil de establecer restricciones de rango es utilizar el método *SetRange*, que ya hemos visto en acción.

```
void __fastcall TTable::SetRange(
    const TVarRec *StartValues, const int StartValues_Size,
    const TVarRec *EndValues, const int EndValues_Size);
```

La aplicación de este método provoca la actualización inmediata de la tabla. El método inverso a la aplicación de un rango es la cancelación del mismo mediante una llamada a *CancelRange*:

```
void __fastcall TTable::CancelRange();
```

En alguna que otra ocasión puede ser útil la propiedad *KeyExclusive*. Si esta propiedad tiene el valor *True*, los extremos del rango son descartados; normalmente la propiedad vale *False*.

El ejemplo de rangos de casi todos los libros

Si el índice activo de una tabla está definido sobre una columna alfanumérica, se puede realizar una sencilla demostración de la técnica de rangos. Reconozco, no obstante, que no es un ejemplo muy original, pues casi todos los libros de C++ Builder traen alguna variante del mismo, pero no he encontrado todavía algo mejor.

Para esta aplicación utilizaremos un formulario típico de exploración con una rejilla, mediante el cual visualizaremos una tabla ordenada por alguna columna de tipo cadena. He elegido la tabla de clientes de *bcdemos*, *customers.db*, que tiene un índice *ByCompany* para ordenar por nombre de compañía:

Objeto	Propiedad	Valor
<i>Form1</i>		(El formulario principal)
<i>Table1</i>		(La tabla de clientes)
	<i>DatabaseName</i>	<i>bcdemos</i>
	<i>TableName</i>	<i>customer.db</i>
	<i>IndexName</i>	<i>ByCompany</i>
<i>DataSource1</i>		(Fuente de datos para <i>Table1</i>)
	<i>DataSet</i>	<i>Table1</i>
<i>DBGrid1</i>		(La rejilla de datos)
	<i>DataSource</i>	<i>DataSource1</i>

Hasta aquí, lo típico. Ahora necesitamos añadir un nuevo control de tipo *TabControl*, de la página *Win32* de la Paleta, para situar un conjunto de pestañas debajo de la rejilla. En este componente modificamos la propiedad *Tabs*, que determina el conjunto de pestañas del control. Esta propiedad, que es una lista de cadenas de caracteres, debe tener 27 líneas. La primera línea debe contener un asterisco, o la palabra *Todos*, o lo que más le apetezca. Las restantes 26 deben corresponder a las 26 letras del alfabeto (no conozco a nadie cuyo nombre comience con Ñ, al menos en España): la primera será una A, la segunda una B, y así sucesivamente. Después asigne las siguientes propiedades:

Propiedad	Valor
<i>Align</i>	<i>alBottom</i>
<i>TabPosition</i>	<i>tpBottom</i>
<i>TabWidth</i>	18
<i>Height</i>	24

Cuando tenga lista la interfaz de usuario, realice una doble pulsación sobre el control para interceptar su evento *OnChange*:

```

void __fastcall TForm1::TabControllChange(TObject *Sender)
{
    AnsiString Letra;
    Letra = TabControll->Tabs->Strings[TabControll->TabIndex];
    if (TabControll->TabIndex == 0)
        // Si es la pestaña con el asterisco ...
        // ... mostramos todas las filas.
        Table1->CancelRange();
    else
        // Activamos el rango correspondiente
        Table1->SetRange(
            ARRAYOFCONST((Letra)),
            ARRAYOFCONST((Letra + "zzz")));
    // Actualizamos los controles asociados
    Table1->Refresh();
}

```

Aunque la actualización de los controles asociados a la tabla ocurre automáticamente en la mayoría de las situaciones, hay casos en los cuales es imprescindible la llamada al método *Refresh*.

Código	Apellidos	Nombre	Extensión	Contrato	Salario
34	Baldwin	Janet	2	21/03/91	\$23.300,00
105	Bender	Oliver H.	255	8/10/92	\$36.799,00
28	Bennet	Ann	5	1/02/91	\$34.482,80
83	Bishop	Dana	290	1/06/92	\$45.000,00
109	Brown	Kelly	202	4/02/93	\$27.000,00
71	Burbank	Jennifer M.	289	15/04/92	\$45.332,00

Más problemas con los índices de dBase

dBase nos sigue dando problemas con sus famosos índices de expresiones; nuevamente tenemos dificultades con la asignación automática de valores a campos en el método *SetRange*. Para solucionar este inconveniente, hay que descomponer la llamada a *SetRange* en las funciones de más bajo nivel *SetRangeStart*, *SetRangeEnd* y *ApplyRange*:

```

void __fastcall TTable::SetRangeStart();
void __fastcall TTable::SetRangeEnd();
void __fastcall TTable::ApplyRange();

```

Los métodos *SetRangeStart* y *SetRangeEnd* indican que las asignaciones a campos que se produzcan a continuación especifican, en realidad, los valores mínimo y máximo del rango; el rango se activa al llamar al método *ApplyRange*. El ejemplo del epígrafe anterior puede escribirse de la siguiente forma mediante estos métodos:

```

Table1->SetRangeStart();
Table1->FieldValues["Company"] = Letra;
Table1->SetRangeEnd();
Table1->FieldValues["Company"] = Letra + "zzz";
Table1->ApplyRange();

```

Como es lógico, si el índice activo para la tabla o el criterio de ordenación establecido en *IndexFieldNames* contemplan varias columnas, hay que realizar varias asignaciones después de *SetRangeStart* y *SetRangeEnd*, una por cada columna del índice o del criterio. Por ejemplo, si *Table1* se refiere a la tabla *employees.db*, y está activo el índice *ByName*, definido sobre los apellidos y el nombre de los empleados, es posible establecer un rango compuesto con los empleados del siguiente modo:

```

Table1->SetRangeStart();
Table1->FieldValues["LastName"] = "A";
Table1->FieldValues["FirstName"] = "A";
Table1->SetRangeEnd();
Table1->FieldValues["LastName"] = "Azzz";
Table1->FieldValues["FirstName"] = "Azzz";
Table1->ApplyRange();

```

El ejemplo anterior sería, precisamente, lo que tendríamos que hacer si la tabla en cuestión estuviese en formato *dbf* y tuviéramos como índice activo un índice de expresiones definido por la concatenación del apellido y del nombre. Recuerde, no obstante, que este rango compuesto no afecta de forma independiente a los dos campos involucrados, y que desde un punto de vista práctico tiene poca utilidad.

Del mismo modo que contamos con los métodos *SetKey* y *EditKey* para la búsqueda por índices, tenemos también los métodos *EditRangeStart* y *EditRangeEnd*, como variantes de *SetRangeStart* y *SetRangeEnd* cuando el índice está definido sobre varias columnas. Estos dos nuevos métodos permiten la modificación del *buffer* que contiene los valores extremos del rango sin borrar los valores almacenados con anterioridad. En el ejemplo anterior, si quisiéramos modificar el rango de modo que incluya solamente a los apellidos que comienzan con la letra B, dejando intacta la restricción de que los nombres empiecen con A, podríamos utilizar el siguiente código:

```

Table1->EditRangeStart();
Table1->FieldValues["LastName"] = "B";
Table1->EditRangeEnd();
Table1->FieldValues["LastName"] = "Bzzz";
Table1->ApplyRange();

```

Cómo crear un índice temporal

Para añadir un nuevo índice a los existentes desde un programa escrito en C++ Builder, hay que utilizar el método *AddIndex*, del componente *TTable*, que debe ejecutarse

sobre una tabla abierta en exclusiva. Los parámetros de este método son similares a los del método *Add* de la clase *TIndexDef*:

```
void __fastcall TTable::AddIndex(const AnsiString Nombre,
    const AnsiString Definicion, TIndexOptions Opciones,
    const AnsiString CamposDescendientes);
```

La *Definicion*, en la mayoría de los casos, es la lista de campos separados por puntos y comas. Pero si la tabla está en formato dBase, podemos utilizar la opción *ixExpression* en el conjunto de opciones, y especificar una expresión en el parámetro *Definicion*:

```
TablaDBase->AddIndex("PorNombre", "NOMBRE+APELLIDOS",
    TIndexOptions() << ixExpression, "");
```

Usted se estará preguntando para qué demonios sirve el último parámetro del método. Yo también. Este parámetro se ha colado sigilosamente en la versión 4 de la VCL, pero es curioso que en Delphi 4 permite un valor por omisión (la cadena vacía, por supuesto), mientras que C++ Builder (¡que ha tenido parámetros por omisión desde siempre!) no lo permite.

En realidad, esta es una característica de Paradox nivel 7; no lo intente sobre las tablas del alias *bedemos*, pues casi todas están en el formato de Paradox 5. Supongamos que una tabla tiene los campos *Pais*, *Ciudad* y *Ventas*. Queremos ordenar primero por países, pero dentro de cada país necesitamos ordenar descendentemente de acuerdo al total de ventas. Lo que tenemos que hacer es pasar los campos descendentes, en este caso uno solo, en el último parámetro de *AddIndex*:

```
Table1->AddIndex("PaisVentas", "Pais;Ventas",
    TIndexOptions() << ixCaseInsensitive, "Ventas");
```

¿A que no adivina qué formato de bases de datos presenta problemas durante la creación de índices? ¡Claro está, dBase! Los desarrolladores de la VCL pasaron por alto una nueva opción del BDE durante la creación de índices para dBase: se puede pedir que un índice sea único (*unique*) o distinto (*distinct*). Un índice único rechaza cualquier intento de inserción de un registro con clave duplicada. Un índice distinto sencillamente reemplaza la clave anterior; esto no tiene mucho sentido, pero existe por razones históricas. Y, por supuesto, está también el índice que permite entradas repetidas. En total, tres tipos de índices. Y una sola opción, *ixUnique*, para especificar la condición de unicidad. Como resultado, los índices dBase creados desde C++ Builder no pueden ser *únicos*. Tampoco pueden serlo los creados mediante *Database Desktop*. Tendremos que esperar al capítulo sobre el API del BDE para aprender a crear índices únicos para dBase nosotros mismos.

Para eliminar el índice recién creado necesitamos el método *DeleteIndex*. Para que este procedimiento pueda cumplir su tarea es necesario que la tabla esté abierta en modo exclusivo. Esta es la declaración de *DeleteIndex*:

```
void __fastcall TTable::DeleteIndex(const AnsiString Nombre);
```

Aunque *AddIndex* y *DeleteIndex* funcionan para tablas pertenecientes a bases de datos SQL, es preferible utilizar instrucciones SQL lanzadas desde componentes *TQuery* para crear índices desde un programa. Vea el capítulo 27 para más información.

Métodos de búsqueda

EN EL CAPÍTULO ANTERIOR estudiamos la búsqueda de valores utilizando índices y el uso de rangos como forma de restringir las filas accesibles de una tabla. En este capítulo estudiaremos los restantes métodos de búsqueda y filtrado que ofrece C++ Builder. Comenzaremos con los filtros, un método de especificación de subconjuntos de datos, y luego veremos métodos de búsqueda directa similares en cierta forma a *FindKey* y *FindNearest*, pero más generales.

Filtros

Los filtros nos permiten limitar las filas visibles de un conjunto de datos mediante una condición arbitraria establecida por el programador. De cierta manera, son similares a los rangos, pero ofrecen mayor flexibilidad y generalidad, pues no están limitados a condiciones sobre las columnas del índice activo. Cuando se aplican a tablas locales, son menos eficientes, pues necesitan ser evaluados para cada fila del conjunto de datos original. En cambio, rangos y filtros se implementan en cliente/servidor por medio de mecanismos similares, al menos cuando hablamos de filtros definidos por expresiones, y en el caso típico nos ofrecen la misma velocidad.

En la VCL 1, donde no existían filtros, había que utilizar consultas SQL para simular este recurso. En efecto, una tabla filtrada es equivalente a una consulta sobre la misma tabla con la condición del filtro situada en la cláusula **where**. No obstante, en muchos casos el uso de los filtros es preferible al uso de consultas, si nos andamos con cuidado, sobre todo por su mayor dinamismo.

Existen dos formas principales de establecer un filtro en C++ Builder. La más general consiste en utilizar el evento *OnFilterRecord*; de esta técnica hablaremos más adelante. La otra forma, más fácil, es hacer uso de la propiedad *Filter*. Estas son las propiedades relacionadas con el uso de filtros:

Propiedad	Significado
<i>AnsiString Filter</i>	Contiene la condición lógica de filtrado
bool Filtered	Indica si el filtro está “activo” o “latente”

Propiedad	Significado
<i>TFilterOptions FilterOptions</i>	Opciones de filtrado; las posibles opciones son <i>foCaseInsensitive</i> y <i>foNoPartialCompare</i> .

La propiedad *Filtered*, de tipo lógico, determina si tiene lugar la selección según el filtro o no; más adelante veremos cómo podemos aprovechar el filtro incluso cuando no está activo.

La condición de selección se asigna, como una cadena de caracteres, en la propiedad *Filter*. La sintaxis de las expresiones que podemos asignar en esta propiedad es bastante sencilla; básicamente, se reduce a comparaciones entre campos y constantes, enlazadas por operadores *and*, *or* y *not*. Por ejemplo, las siguientes expresiones son válidas:

```
Pais = 'Siam'
(Provincia <> '') or (UltimaFactura > '4/07/96')
Salario >= 30000 and Salario <= 100000
```

Si el nombre del campo contiene espacios o caracteres especiales, hay que encerrar el nombre del campo entre corchetes:

```
[Año] = 1776
```

Cuando se trata de tablas de Paradox y dBase, siempre hay que comparar el valor del campo con una constante; no se pueden comparar los valores de dos campos entre sí. En contraste, esta limitación no existe cuando se utilizan tablas pertenecientes a bases de datos cliente/servidor.

Esto no lo dice la documentación...

De no ser por esos “detalles” que se le olvidan a los escritores de manuales, mal lo pasaríamos los escritores de libros. Con el tema de los filtros, al equipo de documentación de C++ Builder se le quedaron un par de trucos en el tintero. El primero de ellos tiene que ver con la posibilidad de utilizar algo parecido al operador **is null** de SQL en una expresión de filtro. Por ejemplo, si queremos filtrar de la tabla de clientes, *customer.db*, solamente aquellas filas que tengan una segunda línea de dirección no nula, la columna *Addr2*, la expresión apropiada es:

```
Addr2 <> NULL
```

Con la constante *Null* solamente podemos comparar en busca de igualdades y desigualdades. Si, por el contrario, queremos los clientes con la segunda línea de dirección no nula, con toda naturalidad utilizamos esta otra expresión:

```
Addr2 = NULL
```

Recuerde que **null** no es exactamente lo mismo que una cadena vacía, aunque en Paradox se representa de esta manera.

El segundo de los trucos no documentados está relacionado con las búsquedas parciales. Por omisión, C++ Builder activa las búsquedas parciales dentro de las tablas cuando no se especifica la opción *foNoPartialCompare* dentro de la propiedad *FilterOptions*. Pero si asignamos la siguiente expresión a la propiedad *Filter* de la tabla de clientes, no logramos ninguna fila:

```
Company = 'A'
```

Esto fue lo que se les olvidó aclarar: hay que terminar la constante de cadena con un asterisco. La expresión correcta es la siguiente:

```
Company = 'A*'
```

De esta manera obtenemos todas las compañías cuyos nombres comienzan con la letra *A*. Sin embargo, por razones que explicaré en el capítulo 25, que se ocupa de la comunicación cliente/servidor, prefiero sustituir la expresión anterior por esta otra:

```
Company >= 'A' and Company < 'B'
```

Un ejemplo con filtros rápidos

Es fácil diseñar un mecanismo general para la aplicación de filtros por el usuario de una rejilla de datos. La clave del asunto consiste en restringir el conjunto de datos de acuerdo al valor de la celda seleccionada en la rejilla. Si tenemos seleccionada, en la columna *Provincia*, una celda con el valor *Madrid*, podemos seleccionar todos los registros cuyo valor para esa columna sea igual o diferente de Madrid. Si está seleccionada la columna *Edad*, en una celda con el valor 30, se puede restringir la tabla a las filas con valores iguales, mayores o menores que este valor. Pero también queremos que estas restricciones sean *acumulativas*. Esto significa que después de limitar la visualización a los clientes de Madrid, podamos entonces seleccionar los clientes con más de 30 años que viven en Madrid. Y necesitamos poder eliminar todas las condiciones de filtrado.

Por lo tanto, comenzamos con la ficha clásica de consulta: una rejilla de datos y una barra de navegación conectada a una tabla simple; si quiere experimentar, le recomiendo conectar la rejilla a la tabla *customer* del alias *bcdemo*, que tiene columnas de varios tipos diferentes. A esta ficha básica le añadimos un menú emergente, *PopupMenu1*, que se conecta a la propiedad *PopupMenu* de la rejilla; basta con esto para que el menú se despliegue al pulsar el botón derecho del ratón sobre la rejilla.

Para el menú desplegable especificamos las siguientes opciones:

Comando de menú	Nombre del objeto de menú
Igual a (=)	<i>miIgual</i>
Distinto de (<>)	<i>miDistinto</i>
Mayor o igual (>=)	<i>miMayorIgual</i>
Menor o igual (<=)	<i>miMenorIgual</i>
Activar filtro	<i>miActivarFiltro</i>
Eliminar filtro	<i>miEliminarFiltro</i>

Observe que he utilizado las relaciones mayor o igual y menor igual en lugar de las comparaciones estrictas; la razón es que las condiciones individuales se van a conectar entre sí mediante conjunciones lógicas, el operador *and*, y las comparaciones estrictas pueden lograrse mediante una combinación de las presentes. De todos modos, es algo trivial aumentar el menú y el código correspondiente con estas relaciones estrictas.



Ahora debemos crear un manejador de evento compartido por las cuatro primeras opciones del menú:

```

void __fastcall TForm1::Filtrar(TObject *Sender)
{
    Set<TFieldType, ftUnknown, ftDataSet> TiposConComillas;
    TiposConComillas << ftString << ftDate << ftTime << ftDateTime;

    AnsiString Operador, Valor;
    Operador = Sender == miIgual ? "=" :
        Sender == miMayorIgual ? ">=" :
        Sender == miMenorIgual ? "<=" :
        "<>";

    // Extraer el nombre del campo
    AnsiString Campo = DBGrid1->SelectedField->FieldName;
    // Extraer y dar formato al valor seleccionado
    if (TiposConComillas.Contains(DBGrid1->SelectedField->DataType))
        Valor = QuotedStr(DBGrid1->SelectedField->AsString);
    else
    {
        Valor = DBGrid1->SelectedField->AsString;
    }
}

```

```

    for (int i = 1; i <= Valor.Length(); i++)
        if (Valor[i] == DecimalSeparator) Valor[i] = '.';
}
// Combinar la nueva condición con las anteriores
if (Table1->Filter == "")
    Table1->Filter = Format("[%s] %s %s",
        ARRAYOFCONST((Campo, Operador, Valor)));
else
    Table1->Filter = Format("%s AND [%s] %s %s",
        ARRAYOFCONST((Table1->Filter, Campo, Operador, Valor)));
// Activar directamente el filtro
miActivarFiltro->Checked = True;
Table1->Filtered = True;
Table1->Refresh();
}

```

El nombre del campo ha sido encerrado automáticamente entre corchetes, para evitar sorpresas con espacios, acentos y eñes. La parte más trabajosa del método es la que tiene que ver con el formato del valor. Si la columna *Company* tiene el valor *Marteens' Diving Academy*, un filtro por igualdad sobre este valor tendría el siguiente aspecto:

```
Company = 'Marteens' Diving Academy'
```

Tome nota de los apóstrofes repetidos dentro de la constante de cadena; esta es una convención léxica heredada de Pascal. Si, por el contrario, creamos un filtro sobre salarios, no son necesarios los apóstrofes para encerrar el valor:

```
Salary = 100000
```

La función *QuotedStr* nos ayuda a dar formato a una cadena de caracteres, y está definida en la unidad *SysUtils*. También hay que tener cuidado con el formato de los campos numéricos con decimales. Nuestro separador de decimales es la coma, mientras que la VCL espera un punto.

En C++ Builder 4, todos estos problemas se resuelven fácilmente, pues se pueden encerrar entre comillas todos los valores constantes, incluidos los numéricos. De este modo, para todos los tipos de datos le daríamos formato a la constante mediante la función *QuotedStr*.

En el método que añade la nueva condición al filtro, se ha activado de paso el filtro, estableciendo la propiedad *Filtered* de la tabla a *True*; de esta forma se obtiene inmediatamente una idea de lo que estamos haciendo. Independientemente de lo anterior, es conveniente tener una opción para activar y desactivar manualmente el filtro, y de esto se encarga el comando de menú *Activar filtro*:

```

void __fastcall TForm1::miActivarFiltroClick(TObject *Sender)
{
    miActivarFiltro->Checked = ! miActivarFiltro->Checked;
}

```

```

// Activar o desactivar en dependencia ...
// ... del estado de la opción del menú.
Table1->Filtered = miActivarFiltro->Checked;
Table1->Refresh();
}

```

Por último, se requiere un comando para eliminar todas las condiciones establecidas:

```

void __fastcall TForm1::miEliminarFiltroClick(TObject *Sender)
{
    miActivarFiltro->Checked = False;
    Table1->Filtered = False;
    Table1->Filter = "";
    Table1->Refresh();
}

```

El evento *OnFilterRecord*

Más posibilidades ofrece la interceptación del evento *OnFilterRecord*. Este es el evento típico que en su encabezamiento tiene un parámetro lógico pasado por referencia, para que aceptemos o rechacemos el registro actual:

```

void __fastcall TForm1::Table1FilterRecord(TDataSet *Sender,
    bool &Accept);

```

El parámetro *Accept* trae por omisión el valor *True*; solamente si queremos rechazar un registro necesitamos asignarle *False* a este parámetro. El algoritmo de decisión que empleemos, por otra parte, puede ser totalmente arbitrario; debemos recordar, sin embargo, que este evento se disparará para cada fila de la tabla original, por lo cual el algoritmo de filtrado debe ser lo más breve y eficiente posible.

¿Cómo podemos aprovechar este evento? En primer lugar, se puede utilizarlo para expresar relaciones que no sean simples comparaciones. Por ejemplo, quiero seleccionar solamente las compañías que pertenezcan al grupo multinacional Marteens (la fantasía no tributa todavía a Hacienda, ¿no?). Una posible solución es utilizar este manejador de eventos:

```

void __fastcall TForm1::Table1FilterRecord(TDataSet *Sender,
    bool &Accept)
{
    AnsiString Company = Sender->FieldValues["Company"];
    Accept = Company.UpperCase.Pos("MARTEENS") != 0;
}

```

El método *UpperCase* lleva una cadena a mayúsculas y *Pos* busca un subcadena dentro de otra. Este otro ejemplo compara entre sí los prefijos de los números de teléfono y de fax:

```

void __fastcall TForm1::Table1FilterRecord(TDataSet *Sender,
    bool &Accept)
{
    AnsiString Phone = Sender->FieldValues["Phone"];
    AnsiString Fax = Sender->FieldValues["Fax"];
    Accept = Phone.SubString(1, 3) != Fax.SubString(1, 3);
    // ¡Estos tienen el teléfono y el fax en distintas ciudades!
}

```

También se puede aprovechar el filtro para comparar entre sí dos campos de una tabla Paradox y dBase.

Medite bien antes de decidirse a utilizar *OnFilterRecord*. Tenga por seguro que este filtro se aplica en el cliente, lo cual implica que la aplicación debe bajarse a través de la red incluso los registros que no satisfacen al filtro. Todo depende, sin embargo, del nivel de selectividad que esperamos de la expresión.

Localización y búsqueda

Si el uso de filtros es similar al uso de rangos, en cierto sentido, los métodos *Locate* y *Lookup* amplían las posibilidades de los métodos de búsqueda con índices. El primero de estos dos métodos localiza la primera fila de una tabla que tenga cierto valor almacenado en una columna, sin importar si existe o no un índice sobre dicha columna. La declaración de este método es:

```

bool __fastcall TDataSet::Locate(const AnsiString Columnas,
    const Variant &Valores, TLocateOptions Opciones);

```

En el primer parámetro se pasa una lista de nombres de columnas; el formato de esta lista es similar al que hemos encontrado en la propiedad *IndexFieldNames*: las columnas se separan entre sí por puntos y comas. Para cada columna especificada hay que suministrar un valor. Si se busca por una sola columna, necesitamos un solo valor, el cual puede pasarse directamente, por ser el segundo parámetro de tipo *Variant*. Si se especifican dos o más columnas, tenemos que pasar una matriz variante; en breve veremos ejemplos de estas situaciones. Por último, el conjunto de opciones del tercer parámetro puede incluir las siguientes:

Opción	Propósito
<i>loCaseInsensitive</i>	Ignorar mayúsculas y minúsculas
<i>loPartialKey</i>	Permitir búsquedas parciales en columnas alfanuméricas

Cuando *Locate* puede encontrar una fila con los valores deseados en las columnas apropiadas, devuelve *True* como resultado, y cambia la fila activa de la tabla. Si, por el contrario, no se localiza una fila con tales características, la función devuelve *False* y no se altera la posición del cursor sobre la tabla. El algoritmo de búsqueda imple-

mentado para *Locate* es capaz de aprovechar los índices existentes. Si el conjunto de columnas no puede aprovechar un índice, *Locate* realiza la búsqueda mediante filtros. Esto es lo que dice la documentación de Inprise/Borland; en el capítulo 25 veremos cómo se implementan realmente estas operaciones en bases de datos SQL.

Ahora veamos un par de ejemplos sencillos de traspaso de parámetros con *Locate*. El uso más elemental de *Locate* es la localización de una fila dado el valor de una de sus columnas. Digamos:

```
if (! tbClientes->Locate("CustNo",
    tbPedidos->FieldValues["CustNo"], TLocateOptions())
    ShowMessage("Se nos ha extraviado un cliente en el bosque...");
```

En este caso, el valor a buscar ha sido pasado directamente como un valor variante, pero podíamos haber utilizado un entero con el mismo éxito, suponiendo que el campo *Código* es de tipo numérico:

```
if (! tbClientes->Locate("CustNo", 007, TLocateOptions())
    ShowMessage("... o se lo ha tragado la tierra");
```

Se puede aprovechar este algoritmo para crear un campo calculado sobre la tabla de clientes, que diga si el cliente ha realizado compras o no. Suponiendo que el nuevo campo, *HaComprado*, es de tipo lógico, necesitamos la siguiente respuesta al evento *OnCalcFields* de la tabla *TablaClientes*:

```
void __fastcall TForm1::tbClientesCalcFields(TDataSet *TDataSet)
{
    tbClientes->FieldValues["HaComprado"] = tbPedidos->Locate(
        "CustNo", tbClientes->FieldValues["CustNo"],
        TLocateOptions());
}
```

Consideremos ahora que queremos localizar un empleado, dados el nombre y el apellido. La instrucción necesaria es la siguiente:

```
tbEmpleados->Locate("LastName;FirstName",
    VarArrayOf(ARRAYOFCONST((Apellido, Nombre))), TLocateOptions());
```

En primer término, hay que mencionar los nombres de ambas columnas en el primer parámetro, separadas por punto y coma. Después, hay que pasar los dos valores correspondientes como una matriz variante; la función *VarArrayOf* es útil para esta última misión. En este ejemplo, las variables *Apellido* y *Nombre* son ambas de tipo *AnsiString*, pero pueden pertenecer a tipos diferentes, en el caso más general.

Cuando la búsqueda se realiza con el objetivo de recuperar el valor de otra columna del mismo registro, se puede aprovechar el método *Lookup*:

```
Variant TDataSet::Lookup(const AnsiString Columns,
                        const Variant Valores, const AnsiString ColResultados);
```

Lookup realiza primero un *Locate*, utilizando los dos primeros parámetros. Si no se puede encontrar la fila correspondiente, *Lookup* devuelve el valor variante especial *Null*; por supuesto, la fila activa no cambia. Por el contrario, si se localiza la fila adecuada, la función extrae los valores de las columnas especificadas en el tercer parámetro. Si se ha especificado una sola columna, ese valor se devuelve en forma de variante; si se especificaron varias columnas, se devuelve una matriz variante formada a partir de estos valores. A diferencia de *Locate*, en este caso no se cambia la fila activa original de la tabla al terminar la ejecución del método.

Por ejemplo, la siguiente función localiza el nombre de un cliente, dado su código:

```
AnsiString TDataModule1::ObtenerNombre(int Codigo)
{
    return VarToStr(tbClientes->Lookup("Código", Codigo,
                                       "Compañía"));
}
```

He utilizado la función *VarToStr* para garantizar la obtención de una cadena de caracteres, aún cuando no se encuentre el código de la compañía; en tal situación, *Lookup* devuelve el variante *Null*, que es convertido por *VarToStr* en una cadena vacía.

También se puede utilizar *Lookup* eficientemente para localizar un código de empleado dado el nombre y el apellido del mismo:

```
int TDataModule1::ObtenerCodigo(const AnsiString Apellido,
                               const AnsiString Nombre)
{
    Variant V = tbEmpleados->Lookup("Apellido;Nombre",
                                   VarArrayOf(ARRAYOFCONST((Apellido, Nombre))), "Código");
    if (VarIsNull(V))
        DatabaseError("Empleado no encontrado", 0);
    return V;
}
```

Por variar, he utilizado una excepción para indicar el fallo de la búsqueda; esto equivale a asumir que lo normal es que la función *ObtenerCodigo* deba encontrar el registro del empleado. Note nuevamente el uso de la función *VarArrayOf*, además del uso de *VarIsNull* para controlar la presencia del valor variante nulo.

Por último, presentaremos la función inversa a la anterior: queremos el nombre completo del empleado dado su código. En este caso, necesitamos especificar dos columnas en el tercer parámetro de la función. He aquí una posible implementación:

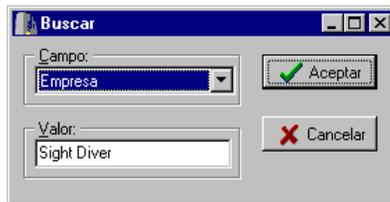
```

AnsiString TDataModule1::NombreDeEmpleado(int Codigo)
{
    Variant V = tbEmpleados->Lookup("Codigo", Codigo,
        "Nombre;Apellido");
    if (VarIsNull(V))
        DatabaseError("Empleado no encontrado", 0);
    return V.GetElement(0) + " " + V.GetElement(1);
}

```

Un diálogo genérico de localización

Se puede programar un diálogo general de búsqueda que aproveche el método *Locate* para localizar la primera fila de una tabla que tenga determinado valor en determinada columna. Este diálogo genérico se programa una sola vez y puede utilizarse sobre cualquier tabla en la que se quiera hacer la búsqueda.



Necesitamos un formulario, al que denominaremos *TDlgBusqueda*, con un combo, llamémosle *cbColumnas*, de estilo *csDropDownList*, un cuadro de edición, de nombre *edValor*, y el par típico de botones para aceptar o cancelar el diálogo. Definiremos también, de forma manual, los siguientes métodos y atributos en la declaración de clase del formulario:

```

class TDlgBusqueda : public TForm
{
    // ...
private:
    TTable *FTabla;
    void __fastcall AsignarTabla(TTable *Valor);
protected:
    __property TTable *Tabla = {read=FTabla, write=AsignarTabla};
public:
    bool __fastcall Ejecutar(TTable *ATable);
}

```

El atributo *FTabla* servirá para recordar la última tabla utilizada para la búsqueda. Para llenar el combo de columnas con los nombres de los campos de la tabla sobre la cual se realiza la búsqueda, se utiliza el método *AsignarTabla*:

```

void __fastcall TDlgBusqueda::AsignarTabla(TTable *Valor)
{
    if (Valor != FTabla)

```

```

    {
        cbColumnas->Items->Clear();
        edValor->Text = "";
        for (int i = 0; i < Valor->FieldCount; i++)
        {
            TField *f = Valor->Fields->Fields[i];
            if (f->FieldKind == fkData
                && ! ftNonTextTypes.Contains(f->DataType))
                cbColumnas->Items->AddObject(
                    f->DisplayLabel, Valor->Fields->Fields[i]);
        }
        cbColumnas->ItemIndex = 0;
        FTabla = Valor;
    }
}

```

Este método solamente modifica los valores del combo si la tabla parámetro es diferente de la última tabla asignada; así se ahorra tiempo en la preparación de la búsqueda. El algoritmo también verifica que los campos a añadir no sean campos calculados o de referencia, con los que el método *Locate* no trabaja. Por las mismas razones se excluyen los campos BLOB de la lista de columnas; la constante de conjunto *ftNonTextTypes* está definida en la unidad *DB*. Finalmente, se añaden las etiquetas de visualización, *DisplayLabel*, en vez de los nombres originales de campos. Para poder encontrar el campo original sin tener que efectuar una búsqueda de estas etiquetas, la inserción dentro del vector *Items* del combo se realiza mediante el método *AddObject* en vez de *Add*; de esta manera, asociamos a cada elemento del combo el puntero al campo correspondiente.

La búsqueda en sí se realiza en el método *Ejecutar*:

```

bool __fastcall TDlgBusqueda::Ejecutar(TTable *ATable)
{
    bool Rslt = False;
    Tabla = ATable;
    if (ShowModal() == mrOk)
    {
        TField *Campo = (TField*)(cbColumnas->Items->Objects[
            cbColumnas->ItemIndex]);
        Rslt = FTabla->Locate(Campo->FieldName, edValor->Text,
            TLocateOptions());
        if (! Rslt)
            Application->MessageBox("Valor no encontrado",
                "Error", MB_OK | MB_ICONSTOP);
    }
    return Rslt;
}

```

Se asigna la tabla, para llenar si es preciso el combo con los nombres de columnas, y ejecutamos el diálogo con *ShowModal*. Si el usuario pulsa el botón *Aceptar*, recuperamos primeramente el puntero al campo seleccionado mediante la propiedad *Objects* de la lista de elementos del cuadro de combinación. A partir del nombre de este campo y del valor tecleado en el cuadro de edición, se efectúa la búsqueda mediante

el método *Locate*. No he implementado el uso de opciones de búsqueda para no complicar innecesariamente el código de este algoritmo, pero usted puede añadir las sin mayor dificultad.

Hasta aquí lo relacionado con el diseño y programación del diálogo genérico de búsqueda. En cuanto al uso del mismo, es muy sencillo. Suponga que estamos explorando una tabla, *Table1*, sobre una ventana con una rejilla. Colocamos un botón de búsqueda en algún sitio libre de la ventana y programamos la siguiente respuesta a su método *OnClick*:

```
void __fastcall TForm1::bnBusquedaClick(TObject *Sender)
{
    DlgBusqueda->Ejecutar(Table1);
}
```

He supuesto que el formulario *DlgBusqueda* se crea automáticamente al ejecutarse la carga del proyecto.

Filtros latentes

Hay que darle al público lo que el público espera. Si un usuario está acostumbrado a actuar de cierta manera frente a cierto programa, esperará la misma implementación de la técnica en nuestros programas. En este caso, me estoy refiriendo a las técnicas de búsqueda en procesadores de textos. Generalmente, el usuario dispone al menos de un par de comandos: *Buscar* y *Buscar siguiente*; a veces, también hay un *Buscar anterior*. Cuando se ejecuta el comando *Buscar*, el usuario teclea lo que quiere buscar, y este valor es utilizado por las restantes llamadas a *Buscar siguiente* y *Buscar anterior*. Y nuestro problema es que este tipo de interacción es difícil de implementar utilizando el método *Locate*, que solamente nos localiza la primera fila que contiene los valores deseados.

La solución a nuestro problema la tienen, curiosamente, los filtros otra vez. Por lo que sabemos hasta el momento, hace falta activar la propiedad *Filtered* para reducir el conjunto de datos activo según la condición deseada. La novedad consiste en la posibilidad de, teniendo *Filtered* el valor *False*, recorrer a saltos los registros que satisfacen la condición del filtro, para lo cual contamos con las funciones *FindFirst*, *FindLast*, *FindNext* y *FindPrior*.

```
bool TDataSet::FindFirst();
bool TDataSet::FindPrior();
bool TDataSet::FindNext();
bool TDataSet::FindLast();
```

Las cuatro funciones devuelven un valor lógico para indicarnos si la operación fue posible o no. Además, para mayor comodidad, los conjuntos de datos tienen una propiedad *Found*, que almacena el resultado de la última operación sobre filtros.

Se puede adaptar el cuadro de diálogo del ejercicio anterior para poder también establecer filtros adecuados a este tipo de búsqueda. Esto lo haremos definiendo un nuevo método, *Buscar*, para establecer el filtro y buscar el primer registro:

```
bool __fastcall TDlgBusqueda::Buscar(TTable *ATable)
{
    bool Rslt = False;
    AsignarTabla(ATable);
    if (ShowModal() == mrOk)
    {
        TField* Campo = (TField*)(cbColumnas->Items->Objects[
            cbColumnas->ItemIndex]);
        FTabla->Filter = Format("[%s] = %s",
            ARRAYOFCONST((Campo->FieldName,
                QuotedStr(edValor->Text))));
        Rslt = FTabla->FindFirst();
        if (!Rslt)
            Application->MessageBox("Valor no encontrado", "Error",
                MB_ICONERROR | MB_OK);
    }
    return Rslt;
}
```

Sí, aquí tenemos otra vez a nuestra vieja conocida: la función *QuotedStr*, que utilizamos en el ejemplo de filtros rápidos. La implementación del comando *Buscar siguiente* sería algo así:

```
bool __fastcall TDlgBusqueda::BuscarSiguiente()
{
    if (! FTabla->FindNext())
    {
        Application->MessageBox("Valor no encontrado", "Error",
            MB_ICONERROR | MB_OK);
        return False;
    }
    else
        return True;
}
```

Por supuesto, para este tipo de búsqueda es preferible habilitar botones en el propio cuadro de diálogo, para lograr el mayor parecido posible con el cuadro estándar de búsqueda de Windows.

Filter By Example

Este es otro ejemplo de cómo diseñar un mecanismo de búsqueda genérico, que aproveche la técnica de los *filtros latentes*. Vamos a crear un prototipo de ventana que pueda ser aprovechada mediante la herencia visual. En esta ventana podremos situar componentes de edición, uno por cada columna por la que queramos buscar. El filtro estará determinado por los valores que introduzcamos en estos campos de edición, de forma similar a lo que ocurre en el lenguaje de consultas *Query By Example*.

Por ejemplo, supongamos que tenemos una tabla de clientes con campos para el código, el nombre y el teléfono. Entonces la ventana de búsqueda tendrá tres editores, uno por cada campo. Si el usuario teclea estos valores:

Código	>34
Nombre	Micro*
Teléfono	

queremos que la expresión de filtro sea la siguiente:

```
Codigo > '34' and Nombre >= 'Micro' and Nombre < 'Micrp'
```

No se ha generado ninguna comparación para el teléfono, porque el usuario no ha tecleado nada en el campo correspondiente. Observe que he evitado el uso del asterisco al final de la constante, para que el filtro sea realmente eficiente.

Creamos una aplicación, y en su ventana principal situamos una rejilla conectada a una tabla arbitraria. Llamaremos *wndMain* a esta ventana principal. Creamos un nuevo formulario en la aplicación, con el nombre *wndSearch*, y lo guardamos en la unidad *Search*. Lo quitamos de la lista de creación automática, pues cuando lo necesitamos lo crearemos nosotros mismos. Le cambiamos la propiedad *BorderStyle* a *bsToolWindow*, y *FormStyle* al valor *fsStayOnTop*, de modo que siempre se encuentre en primer plano. Luego añadimos una barra de herramientas, con cuatro botones de navegación y, un poco separados, un par de botones, para aplicar el filtro de búsqueda (*bnApply*) y para eliminarlo (*bnClean*).

La idea es que, en los descendientes de este formulario, se añadan cuadros de búsquedas del tipo *TEdit*, es decir, comunes y corrientes. Para asociar un campo de la tabla en que se quiere buscar a cada editor, se asignará el nombre del campo en la propiedad *Text* del control. Por ejemplo, la siguiente figura muestra el aspecto de una ventana de búsqueda sobre la tabla de clientes en tiempo de diseño:



Pero no se preocupe por la propiedad *Text* de estos controles, pues durante la creación de la ventana se utilizará para asociar un campo al control, y después se borrará. Esta tarea es responsabilidad del siguiente método público estático:

```

TwndSearch* TwndSearch::Launch(TMetaClass *aClass, TForm* AOwner,
    TTable *ATable)
{
    TwndSearch *f = NULL;
    LockWindowUpdate(Application->MainForm->ClientHandle);
    try
    {
        for (int i = 0; i < Screen->FormCount; i++)
            if (Screen->Forms[i]->ClassType() == aClass)
                return (TwndSearch*) Screen->Forms[i];
        Application->CreateForm(aClass, &f);
        AOwner->FreeNotification(f);
        f->FTable = ATable;
        for (int i = 0; i < f->ComponentCount; i++)
        {
            TComponent *C = f->Components[i];
            if (dynamic_cast<TEdit*>(C))
            {
                ((TEdit*) C)->Tag =
                    int(ATable->FindField(((TEdit *)C)->Text));
                ((TEdit*) C)->Text = "";
            }
        }
        f->bnApply->Enabled = False;
    }
    finally
    {
        LockWindowUpdate(0);
    }
    return f;
}

```

Como se puede apreciar, se utiliza el valor guardado en el texto del control para buscar el puntero al campo, el cual se asigna entonces a la propiedad *Tag*. La respuesta a los cuatro botones de navegación es elemental:

```

void __fastcall TwndSearch::bnFirstClick(TObject *Sender)
{
    FTable->FindFirst();
}

void __fastcall TwndSearch::bnPriorClick(TObject *Sender)
{
    FTable->FindPrior();
}

```

```

void __fastcall TwndSearch::bnNextClick(TObject *Sender)
{
    FTable->FindNext();
}

void __fastcall TwndSearch::bnLastClick(TObject *Sender)
{
    FTable->FindLast();
}

```

También es predecible la respuesta al botón que elimina el filtro:

```

void __fastcall TwndSearch::bnCleanClick(TObject *Sender)
{
    FTable->Filter = "";
    bnClean->Enabled = False;
    bnApply->Enabled = True;
}

```

Donde realmente hay que teclear duro es en la respuesta al botón que activa el filtro:

```

void __fastcall TwndSearch::bnApplyClick(TObject *Sender)
{
    AnsiString F;
    for (int i = 0; i < ComponentCount; i++)
    {
        TEdit *C = dynamic_cast<TEdit*>(Components[i]);
        if (C != NULL)
            AddFilter(F, C, (TField*)(C->Tag));
    }
    FTable->Filter = F;
    FTable->FindFirst();
    bnApply->Enabled = False;
    bnClean->Enabled = F != "";
}

```

El método *AddFilter* debe haber sido definido en la parte privada de la declaración del formulario, y se encarga de dar el formato correcto al valor tecleado en cada control. Si el campo es de tipo cadena, debe encerrarse el valor entre apóstrofes; si es un número real, hay que sustituir nuestras comas decimales por los puntos americanos:

```

void __fastcall TwndSearch::AddFilter(AnsiString &F, TEdit *E,
TField *AField)
{
    static AnsiString Ops[6] = {"<>", "<=", ">=", "<", ">", "="};

    AnsiString S = E->Text.Trim();
    if (S == "" || AField == NULL) return;
    // Buscar el operador
    AnsiString Op = "=";
    for (int i = 0; i < 6; i++)
        if (S.Pos(Ops[i]) == 1)
            {
                Op = Ops[i];
            }
}

```

```

        S.Delete(1, Op.Length());
        S = S.TrimLeft();
        break;
    }
    // Formatear el valor resultante
    if (Op == "=" && AField->DataType == ftString
        && S.Length() > 1 && S[S.Length()] == '*')
    {
        S.Delete(S.Length(), 1);
        AnsiString S1 = S;
        ++S1[S1.Length()];
        S = Format("[%s]>=%s and [%0:s]<%2:s", ARRAYOFCONST((
            AField->FieldName, QuotedStr(S), QuotedStr(S1))));
    }
    else
        S = "[" + AField->FieldName + "]" + Op + QuotedStr(S);
    // Añadir al filtro existente
    if (F != "")
        AppendStr(F, " AND ");
    AppendStr(F, S);
}

```

Ahora derivamos por herencia visual a partir de esta ventana un nuevo formulario, de nombre *TschClientes*, y añadimos los tres cuadros de edición para los tres campos que queremos que participen en la búsqueda. Recuerde quitarlo de la lista de creación automática. El aspecto en ejecución del diálogo de búsqueda de clientes es el siguiente:



Y la instrucción utilizada para lanzarlo, desde la ventana de exploración principal, es la que mostramos a continuación:

```

void __fastcall TwndMain.Buscar1Click(TObject *Sender)
{
    TschClientes::Launch(__classid(TschClientes), this,
        modDatos->tbCli)->Show();
}

```

Lo principal es que, teniendo esta plantilla, no hace falta escribir una línea de código en las ventanas de búsqueda que heredan de la misma. Se pueden diseñar estrategias de búsqueda aún más sofisticadas. Por ejemplo, se pueden habilitar teclas para que un campo de edición tome el valor real de la fila activa en la ventana. También se puede hacer que esta ventana se pueda aparcar (*dock*) en la barra de tareas de la ventana principal. Todo eso se lo dejo a su paciencia e imaginación.

Búsqueda en una tabla de detalles

Para terminar, supongamos que la tabla sobre la cual se busca es una tabla de detalles, es decir, una tabla que juega el rol de subordinada en una relación *master/detail*. Quizás deseemos buscar todas las ventas de cierto producto, para mostrar la factura correspondiente; el código del producto debe buscarse en la tabla de detalles, pero las filas visibles de esta tabla están limitadas por el rango implícito en su relación con la tabla de pedidos. No queda más remedio que buscar en otro objeto de tabla, que se refiera a la misma tabla física de pedidos, pero que no participe en una relación *master/detail*. Después tendremos que localizar el pedido correspondiente en la tabla de pedidos, para mostrar la factura completa. Ya sabemos la teoría necesaria para todo esto, y solamente tenemos que coordinar las acciones.

Necesitamos dos ventanas, una para mostrar los pedidos y las líneas de detalles, y otra para seleccionar el artículo a buscar. Esta última es la más sencilla, pues situaremos en la misma un par de botones (*Aceptar* y *Cancelar*), una rejilla sólo para lectura, una fuente de datos y una tabla, *tbArticulos*, que haga referencia a la tabla de artículos *parts.db* de la base de datos *bcemos*. Llamaremos a este formulario *dlgSeleccion*.

En el formulario principal, llamémosle *wndPrincipal*, situaremos las siguientes tablas:

Tabla	Propósito
<i>tbPedidos</i>	Trabaja con la tabla <i>orders.db</i> , del alias <i>bcemos</i> .
<i>tbDetalles</i>	Tabla <i>items.db</i> , en relación <i>master/detail</i> con la anterior.
<i>tbBusqueda</i>	Tabla <i>items.db</i> , pero sin relación <i>master/detail</i> .

También necesitaremos rejillas para mostrar las tablas, y un par de botones: *bnBuscar* y *bnSiguiente*, este último con la propiedad *Enabled* a *False*. La respuesta al evento *OnClick* del botón *bnBuscar* es la siguiente:

```
void __fastcall TwndPrincipal::bnBuscarClick(TObject *Sender)
{
    if (dlgSeleccion->ShowModal() == mrOk)
    {
        tbBusqueda->Filter = "PartNo = " +
            VarToStr(dlgSeleccion->tbArticulos->FieldValues["PartNo"]);
        tbBusqueda->FindFirst();
        bnSiguiente->Enabled = True;
        Sincronizar();
    }
}
```

En el método anterior, ejecutamos el diálogo de selección, inicializamos el filtro, buscamos la primera fila que satisface el criterio de búsqueda y sincronizamos la posición de las tablas visibles con el método *Sincronizar*. Este método lo definimos del siguiente modo:

```

void __fastcall TwndPrincipal::Sincronizar()
{
    if (! tbBusqueda->Found)
        bnSiguiete->Enabled = False;
    else
    {
        tbPedidos->Locate("OrderNo",
            tbBusqueda->FieldValues["OrderNo"], TLocateOptions());
        tbDetalles->GotoCurrent(tbBusqueda);
    }
}

```

Se sabe si la última búsqueda tuvo éxito o no consultando la propiedad *Found* de la tabla de búsqueda. En caso afirmativo, se localiza el pedido correspondiente en la tabla de pedidos, y solamente entonces se procede a activar la fila encontrada de la tabla de detalles; observe el uso del método *GotoCurrent* para realizar la sincronización.

La respuesta al botón *bnSiguiete*, teniendo el método anterior programado, es trivial:

```

void __fastcall TwndPrincipal::bnSiguieteClick(TObject *Sender)
{
    tbBusqueda->FindNext();
    Sincronizar();
}

```


Navegación mediante consultas

TODAS LAS OPERACIONES SOBRE BASES DE DATOS QUE hemos estudiado hasta el momento se han basado en el uso de componentes *TTable*. Ahora mostraremos cómo podemos utilizar el componente *TQuery* para navegar sobre el contenido de una tabla, y cómo recuperar información en general desde una base de datos. Aprovecharemos también la ocasión para conocer algunas peculiaridades del dialecto de SQL local implementado en el Motor de Datos de Borland, y para mostrar la herramienta *SQL Builder*, para generar instrucciones SQL de forma visual.

El componente *TQuery* como conjunto de datos

Para enviar instrucciones SQL a la base de datos se utiliza el componente *TQuery*, que se encuentra en la página *Data Access* de la Paleta de Componentes. Desde el punto de vista de la jerarquía de herencia de la VCL, la clase *TQuery* desciende del tipo *TDBDataSet*, por lo que los objetos de consultas son conjuntos de datos. Esto quiere decir que podemos conectar una fuente de datos a un objeto de consultas para mostrar y editar su contenido desde controles de datos, que podemos movernos por sus filas, extraer información de sus campos; en definitiva, que casi todas las operaciones aplicables a las tablas son aplicables a este tipo de componente.

No obstante, un objeto *TQuery* sólo puede tratarse como un conjunto de datos en el caso especial de que la instrucción SQL que contenga sea una consulta. Si la instrucción pertenece al DDL, DCL, o es una de las instrucciones **update**, **insert** ó **delete**, no tiene sentido pensar en el resultado de la ejecución de la instrucción como si fuera un conjunto de datos. En este caso, tenemos métodos especiales para tratar con el componente.

Para una tabla, las propiedades *DatabaseName* y *TableName* determinan, en lo fundamental, el origen de los datos; para una consulta, necesitamos por lo menos asignar valores a *DatabaseName*, la base de datos contra la cual se ejecuta la instrucción, y *SQL*, la lista de cadenas que contiene la instrucción SQL en sí. Es posible omitir el valor de la propiedad *DatabaseName*. Si se omite esta propiedad, hay que especificar dentro de la instrucción SQL a qué base de datos pertenece cada tabla; más adelante

veremos cómo hacerlo. Esta técnica, sin embargo, no es recomendable si queremos trabajar con una base de datos SQL remota, pues en tal situación la evaluación de la consulta la realiza el SQL local del BDE.

Un componente *TQuery* utilizado como conjunto de datos puede hacer uso de la propiedad *Active*, o de los métodos *Open* y *Close*, para abrir y cerrar la consulta. Una vez abierta la consulta, podemos aplicar casi todas las operaciones que son aplicables a tablas; la gran excepción son las operaciones que se implementan mediante índices. No obstante, también pueden aplicarse filtros a la consulta. El acceso a la información de cada campo se logra del mismo modo: se pueden crear campos persistentes en tiempo de diseño, o se puede acceder dinámicamente a los mismos con las propiedades *Fields*, *FieldValues* y con la función *FieldByName*.

¿Quién ejecuta las instrucciones?

Esta es una buena pregunta, pero estoy seguro de que ya intuye la respuesta. En primer lugar, si la petición se dirige a una base de datos local, esto es, si la base de datos asignada a *DatabaseName* se refiere al controlador *STANDARD*, la instrucción es interpretada en la máquina cliente por el denominado SQL Local, perteneciente al BDE. En contraste, si la propiedad *DatabaseName* se refiere a una base de datos remota, estamos ante lo que el BDE llama *passthrough SQL*, es decir, instrucciones SQL que se “pasan” al servidor remoto y que son ejecutadas por el mismo. Este comportamiento por omisión puede modificarse mediante el parámetro *SQLQRYMODE* de la configuración del BDE, aunque es recomendable dejarlo tal como está.

Las cosas se complican cuando se utilizan *consultas heterogéneas*. En este tipo de consultas se mezclan datos de varias bases de datos. Estas consultas son siempre interpretadas por el SQL local. En una consulta heterogénea, los nombres de las bases de datos se indican delante de los nombres de tablas, en la cláusula **from**. Para poder utilizar este recurso, la propiedad *DatabaseName* de la consulta debe estar vacía o hacer referencia a un alias local. Por ejemplo, en la siguiente consulta mezclamos datos provenientes de una tabla de InterBase y de una tabla Paradox. Los nombres de tablas incluyen el alias utilizando la notación *:ALIAS*; y deben encerrarse entre comillas. Claro está, necesitamos sinónimos para las tablas si queremos cualificar posteriormente los nombres de los campos:

```
select E.FULL_NAME, sum(O.ItemsTotal)
from   ":IBLOCAL:EMPLOYEE" E, ":BCDEMOS:ORDERS" O
where  O.EmpNo = E.EMP_NO
group  by E.FULL_NAME
order  by 2 desc
```

Cuando se trata de una tabla local para la cual sabemos el directorio, pero no tenemos un alias, podemos especificar el directorio en sustitución del nombre de alias:

```
select *
from "C:\Data\Orders.db"
```

La propiedad *Local* de la clase *TQuery* indica si la base de datos asociada a la consulta es cliente/servidor o de escritorio.

Consultas actualizables

Como sabemos, existen reglas matemáticas que determinan si una expresión relacional puede considerarse actualizable o no. En la práctica, los sistemas relacionales tienen sus propias reglas para determinar qué subconjunto de todas las expresiones posibles pueden ser actualizadas. Las reglas de actualizabilidad las encontramos cuando se definen vistas en un sistema SQL, y las volvemos a encontrar al establecer consultas sobre una base de datos desde el ordenador cliente. En C++ Builder, para pedir que una consulta retorne un cursor actualizable, de ser posible, es necesario asignar *True* a la propiedad lógica *RequestLive* de la consulta. El valor por omisión de esta propiedad es *False*.

No obstante, *RequestLive* es solamente una petición. El resultado de esta petición hay que extraerlo de la propiedad *CanModify* una vez que se ha activado la consulta. Si la consulta se ejecuta contra una base de datos local y su sintaxis permite la actualización, el BDE retorna un cursor actualizable; en caso contrario, el BDE proporciona un conjunto de datos de sólo lectura. Sin embargo, si la petición se establece contra un servidor SQL, puede llevarse la desagradable sorpresa de provocar una excepción si pide una consulta “viva” y el sistema se la niega.

Desafortunadamente, el algoritmo que decide si una expresión **select** es actualizable o no depende del sistema de base de datos. Aquí exponemos las reglas de actualizabilidad del SQL local; es una regla en dos partes, pues depende de si la instrucción utiliza una sola tabla o utiliza varias. Si se utiliza una sola tabla, deben cumplirse las siguientes restricciones para la actualizabilidad:

- La tabla base debe ser actualizable (...elemental, Watson...).
- No se utilizan **union**, **minus**, **intersect**, **group by** ó **having**.
- Si se utiliza **distinct**, que sea innecesario (!).
- No se permiten funciones de conjuntos en la selección.
- No se permiten subconsultas.
- Si existe un **order by**, que se pueda implementar mediante un índice.

La explicación de la regla 3 era demasiado larga y rompía la simetría de la lista. Quería decir simplemente que si aparece la palabra **distinct** en la cláusula de selección,

deben aparecer también todos los campos de la clave; en ese caso, la instrucción también podría haberse escrito sin especificar **distinct**.

Por otra parte, deben cumplirse las siguientes reglas si la cláusula **from** contiene varias tablas:

- Las tablas están formando todas un encuentro natural (*natural join*), o un encuentro externo (*outer join*) de izquierda a derecha.
- Los encuentros deben implementarse mediante índices.
- No se usa la cláusula **order by**.
- Cada tabla es una tabla base, no una vista.
- Se cumplen las restricciones aplicables de la primera lista.

Curiosamente, las reglas de las consultas actualizables para los servidores SQL dependen no de los servidores, ¡sino del propio BDE! El problema básico consiste en que el BDE siempre implementa los cursores bidireccionales en el cliente a partir de cursores unidireccionales del servidor, aún si el servidor soporta cursores en dos sentidos. Supongamos que hemos leído 50 registros desde el servidor y decidimos regresar al primer registro. El registro activo en el servidor es el número 50, no el número 1. Así que los posibles cambios que hagamos sobre el primer registro no pueden grabarse utilizando sentencias (como **update where current of cursor**) que afecten al registro activo de un cursor del servidor. El BDE se ve obligado a utilizar alguna clave única para establecer la correspondencia entre registro del cliente y registro del servidor. Como resultado, si hay dos o más tablas en la cláusula **from**, la consulta no es actualizable, pues en general no se puede establecer tal correspondencia.

La apertura de un objeto *TQuery* es más rápida que la de un *TTable*, pues el componente de tablas necesita extraer información acerca de los campos, claves, índices y restricciones antes de comenzar a recibir datos del servidor. Este hecho se utiliza como justificación para evitar el uso de tablas. Sin embargo, muchas veces se pasa por alto que una consulta actualizable necesita ejecutar el mismo preámbulo que las tablas, con lo cual los dos componentes tardan lo mismo en abrirse. De todos modos, existe una solución intermedia: utilizar consultas con actualizaciones en caché y objetos de actualización. En el siguiente capítulo estudiaremos más detalles de las técnicas de navegación utilizadas por el BDE.

Existe una forma de permitir modificaciones sobre una consulta no actualizable. Consiste en activar las actualizaciones en caché para el componente *TQuery* y asociarle un objeto de actualización *TUpdateSQL*. Las actualizaciones en caché se controlan mediante la propiedad *CachedUpdates* de los conjuntos de datos, pero se estudiarán mucho más adelante. Los objetos de actualización, por su parte, permiten

especificar reglas por separado para implementar los borrados, modificaciones e inserciones sobre la consulta; estas reglas son también instrucciones SQL.

Siempre hacia adelante

El uso más común de un objeto *TQuery* basado en una instrucción **select** es la visualización de sus datos mediante algún dispositivo de navegación. La navegación nos obliga a una implementación que nos permita movernos arbitrariamente dentro del conjunto de datos producido como resultado de la instrucción SQL. Casi todos los intérpretes SQL construyen una representación física del resultado, que puede consistir, en dependencia de si la consulta es actualizable o no, en una copia física temporal del conjunto de datos generado, o en un fichero de punteros a las filas originales de las tablas implicadas. A este tipo de estructura se le conoce con el nombre de *cursor bidireccional*.

En cambio, si solamente nos tenemos que desplazar por el cursor en una sola dirección, la implementación puede ser menos costosa en algunos casos. Por ejemplo, suponga que la consulta en cuestión consiste en una selección de filas a partir de una sola tabla, como la siguiente:

```
select *
from   Clientes
where  UltimoPedido > "4/7/96"
```

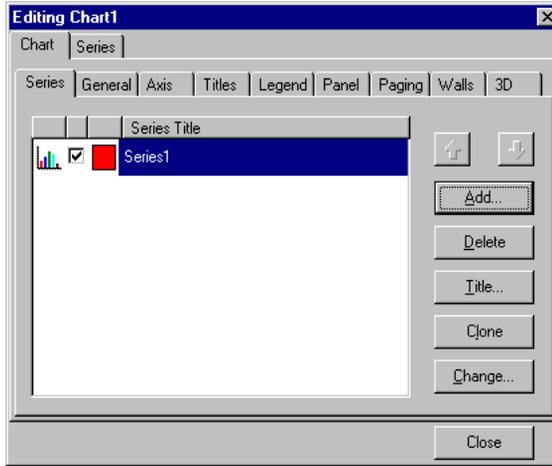
En tal caso, si solamente vamos a desplazarnos hacia adelante al usar esta consulta, el intérprete SQL lo único que tiene que hacer es abrir la tabla base, *Clientes*, e interpretar el método *Next* sobre la consulta como una sucesión de llamadas al método *Next* sobre la tabla base hasta que se cumpla la condición de la cláusula **where**.

Para ilustrar el uso de consultas unidireccionales, le mostraré cómo generar un gráfico de ventas a partir de la información de pedidos. El gráfico que nos interesa debe mostrar la cantidad de clientes por tramos de ventas totales: cuántos clientes nos han comprado hasta \$50.000, cuántos de \$50.001 hasta \$100.000, etc. Por supuesto, necesitamos los totales de ventas por clientes, y esta información podemos extraerla mediante la siguiente consulta:

```
select count(ItemsTotal)
from   Orders
group  by CustNo
```

Esta instrucción se coloca dentro de un objeto *TQuery*, al cual se le modifica a *True* el valor de su propiedad *UniDirectional*.

Para mostrar el gráfico utilizaré el *TTeeChart*, ubicado en la página *Additional* de la Paleta de Componentes. Me estoy adelantando un poco, pues este control será estudiado con más detalles en un capítulo posterior. Por el momento, traiga a un formulario un componente *TTeeChart*, realice un doble clic sobre el mismo y pulse el botón *Add* para añadir una nueva serie al gráfico. Elija entonces un gráfico de barras:



Después necesitamos ir a la página *Series*, y en la página anidada *DataSource* debemos seleccionar *No data*, para que el gráfico aparezca inicialmente vacío. Como resultado de estas acciones, el formulario contiene una nueva variable *Series1*, un puntero a *TBarSeries*, que contendrá los datos del gráfico de barra.

Nos interesa establecer los valores de umbral de la forma más flexible que podemos. Para esto declaramos un método en la definición de tipo del formulario:

```
class TForm1 : public TForm
{
    // ...
private:
    void __fastcall LlenarGrafico(double *Valores, int Valores_High);
};
```

La idea es llamar a este método desde la respuesta al evento *OnCreate* de la ventana:

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    LlenarGrafico(OPENARRAY(double,
        (25000, 50000, 75000, 100000, 150000)));
}
```

Bajo estas suposiciones, el método *LlenarGrafico* se implementa muy fácilmente:

```

void __fastcall TForm1::LlenarGrafico(double *Valores,
int Valores_High)
{
    // Inicializar un vector con los valores
    int* c = new int[Valores_High + 2];
    try {
        memset((void*)c, 0, (Valores_High + 2) * sizeof(int));
        // Abrir la consulta y recorrer sus filas
        Query1->Open();
        try {
            while (! Query1->Eof)
            {
                // Localizar el valor de umbral
                int i = 0;
                while (i <= Valores_High &&
                    Query1->Fields->Fields[0]->AsFloat > Valores[i]) i++;
                c[i]++;
                Query1->Next();
            }
        }
        finally
        {
            Query1->Close();
        }
        for (int i = 0; i <= Valores_High + 1; i++)
            Series1->Add(c[i], "", clTeeColor);
    }
    finally
    {
        delete c;
    }
}

```



Observe que la consulta se examina en una pasada desde el principio hasta el final, sin necesidad de dar marcha atrás. El que realmente signifique algún adelanto establecer un cursor unidireccional o uno bidireccional depende de la implementación del intérprete SQL que ofrezca el sistema de bases de datos.

Consultas paramétricas

Es posible modificar en tiempo de ejecución el contenido de la propiedad *SQL* de una consulta. Esto se realiza en ocasiones para permitir que el usuario determine, casi siempre de una forma visual y más o menos intuitiva, qué información quiere obtener. El programa genera la instrucción SQL correspondiente y ¡zas!, el usuario queda satisfecho. Sin embargo, muchas veces los cambios que distinguen una instrucción de otra se refieren a valores constantes dentro de la expresión. Ahora veremos cómo podemos cambiar de forma más eficiente estos valores sin afectar a la consulta en su totalidad.

Supongamos que el usuario quiere filtrar la tabla de clientes para mostrar los clientes de acuerdo al estado (*State*) al que pertenecen. Traemos a un formulario una rejilla de datos, *DBGrid1*, una fuente de datos, *DataSource1*, un cuadro de edición, *Edit1*, y una consulta *Query1*. Conectamos los componentes de base de datos como es usual, y modificamos las siguientes propiedades del cuadro de edición:

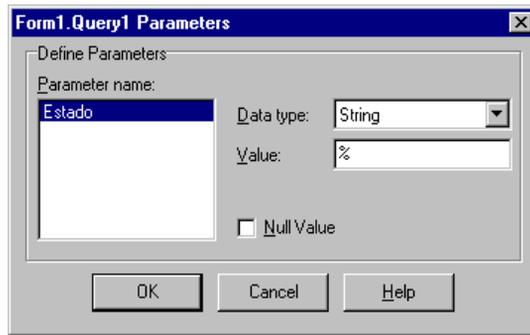
Propiedad	Valor
<i>CharCase</i>	<i>ecUpperCase</i>
<i>MaxLength</i>	2
<i>Text</i>	<Vacío>

Ahora vamos a la consulta. Asignamos *bdemos* a *DatabaseName*, y tecleamos la siguiente instrucción en la propiedad *SQL*:

```
select *
from Customer
where State like :Estado
```

La novedad es el operando *:Estado*, que representa un parámetro de la consulta. Es importante que los dos puntos vayan junto al nombre del parámetro, para que C++ Builder pueda reconocerlo como tal. En una instrucción SQL podemos utilizar tantos parámetros como queramos, y pueden utilizarse como sustitutos de constantes; nunca en el lugar de un nombre de tabla o de columna.

Después de tener el texto de la instrucción, necesitamos asignar un tipo a cada parámetro, e indicar opcionalmente un valor inicial para cada uno de ellos. Esto se hace editando la propiedad *Params*. Más adelante veremos un ejemplo en el cual es necesario no asociar tipos a parámetros, pero esta será la excepción, no la regla. En nuestro ejemplo, asignamos al único parámetro el tipo *String*, y le damos como valor inicial el carácter %; como recordará el lector, si se utiliza un signo de porcentaje como patrón de una expresión **like**, se aceptarán todas las cadenas de caracteres. Una vez que hayamos asignado los parámetros, podemos abrir la consulta, asignando a la propiedad *Active* a *True*.



La imagen anterior corresponde al editor de parámetros de C++ Builder 3. En la versión 4 los parámetros se editan mediante el editor genérico de colecciones.

Para cambiar dinámicamente el valor de un parámetro de una consulta es necesario, en primer lugar, que la consulta esté inactiva. Del mismo modo que sucede con los campos, existen dos formas de acceder al valor de un parámetro: por su posición y por su nombre. Es muy recomendable utilizar el nombre del parámetro, pues es muy probable que si se modifica el texto de la instrucción SQL, la posición de un parámetro varíe también. Para acceder a un parámetro por su nombre, necesitamos la función *ParamByName*:

```
TParam* __fastcall TQuery::ParamByName(const AnsiString Nombre);
```

De todos modos, también es posible utilizar la propiedad *Params*, mediante la cual obtenemos el puntero al parámetro por medio de su posición:

```
__property TParams* Params;
```

En nuestro pequeño ejemplo, la asignación al parámetro debe producirse cuando el usuario modifique el contenido del cuadro de edición. Interceptamos, en consecuencia, el evento *OnChange* del cuadro de edición:

```
void __fastcall TForm1::Edit1Change(TObject *Sender)
{
    Query1->Close();
    try
    {
        Query1->ParamByName("Estado")->AsString = Edit1->Text + "%";
    }
    __finally
    {
        Query1->Open();
    }
}
```

Como se puede apreciar, se garantiza mediante un bloque de protección de recursos la reapertura de la consulta. Utilizamos la función *ParamByName* para acceder al pa-

rámetro; el nombre del parámetro se escribe ahora sin los dos puntos iniciales. Por último, hay que utilizar la propiedad *AsString* (o *AsInteger* o *AsDateTime*...) para manipular el valor del parámetro; no existe una propiedad parecida a *Value*, como en el caso de los campos, para hacer uso del valor sin considerar el tipo del parámetro.

Consultas dependientes

Es posible asociar a un parámetro el valor obtenido de una columna perteneciente a otra tabla o consulta. El cambio de parámetro se produce automáticamente cada vez que cambia la fila en el primer conjunto de datos. De este modo, se puede lograr un efecto similar a las tablas de detalles, pero sobre consultas SQL. A este tipo de consultas se le denomina *consultas dependientes* o, en castellano antiguo: *linked queries*.

Para poder asociar un parámetro de una consulta a una tabla, siga estas instrucciones:

- Asigne a la propiedad *DataSource* de la consulta una fuente de datos enlazada a la tabla maestra. Esta es una propiedad con un nombre inadecuado. Le regalaría una botella de champagne al equipo de desarrolladores de Borland si en la próxima versión de C++ Builder cambiasen el nombre de la propiedad a *MasterSource*, ¡palabra de programador!¹⁶
- No asigne tipo al parámetro en el editor de la propiedad *Params*.

Considere, por ejemplo, la siguiente consulta:

```
select Parts.Description, Items.Qty, Items.Discount
from Items, Parts
where Items.PartNo = Parts.PartNo
```

Mediante esta instrucción podemos obtener un listado de los artículos vendidos, junto a sus descripciones. Esto pudiera servirnos para sustituir a los campos de referencia de C++ Builder si queremos mostrar las descripciones de artículos en una rejilla. Pero la instrucción muestra *todos* los artículos vendidos, mientras que el uso más frecuente de esta información es mostrarla como detalles de los pedidos.

En este caso, la instrucción necesaria es la siguiente:

```
select Parts.Description, Items.Qty, Items.Discount
from Items, Parts
where Items.PartNo = Parts.PartNo
and Items.OrderNo = :OrderNo
```

¹⁶ Me parece que tienen intención de no beberse la botella.

El inconveniente principal de esta técnica, si se utiliza para sustituir los campos de búsqueda de C++ Builder, es que el resultado de la consulta no es actualizable. Le sugiero que pruebe esta consulta con la base de datos *mastsql.gdb*, que se encuentra en el directorio de demostraciones de C++ Builder. El intérprete SQL de InterBase es más eficiente para este tipo de cosas que el SQL local del BDE.

La preparación de la consulta

Si una consulta con parámetros va a ser abierta varias veces, es conveniente *prepararla* antes de su ejecución. Preparar una consulta quiere decir realizar su análisis sintáctico y producir el código de ejecución necesario; por supuesto, cada servidor realiza esta tarea de forma diferente. La preparación de una consulta se realiza por medio del método *Prepare* de la clase *TQuery*. Normalmente, esta operación se realiza automáticamente durante la apertura de la consulta, y la operación inversa tiene lugar cuando se cierra la consulta. Sin embargo, la preparación consume tiempo y recursos. Si los parámetros cambian varias veces durante la vida del objeto de consulta, estaremos repitiendo la misma cantidad de veces la tediosa operación de preparación. La solución es preparar la consulta explícitamente, y deshacer, consecuentemente, la preparación antes de destruir el objeto, o cuando no se vaya a utilizar por mucho tiempo. En especial, si va a utilizar una consulta dependiente, tenga en cuenta que cada vez que se cambie la fila activa de la tabla maestra, se está cerrando y reabriendo la consulta asociada.

Precisamente, la posibilidad de preparar una consulta la primera vez y ejecutarla cuantas veces deseemos, cambiando los parámetros cada vez que haga falta, constituye la ventaja de utilizar parámetros frente a la modificación directa del texto de la consulta.

Si las instrucciones de apertura y cierre de la consulta se realizan explícitamente, la preparación se puede programar de esta manera:

```
void __fastcall TQueryForm::FormCreate(TObject *Sender)
{
    if (! Query1->Prepared)
        Query1->Prepare();
    Query1->Open();
}

void __fastcall TQueryForm::FormClose(TObject *Sender)
{
    Query1->Close();
    if (Query1->Prepared)
        Query1->UnPrepare();
}
```

¿Y qué sucede si hace falta que la consulta esté abierta en tiempo de diseño? El problema es que cuando se dispara el evento *OnCreate* del formulario o del módulo de datos, ya las consultas y tablas que estaban abiertas en tiempo de diseño han sido activadas. Alguien puede tener la idea de realizar la preparación en el evento *BeforeOpen* de estos componentes. Pero no pierda su tiempo: en pruebas realizadas por el autor, la preparación de la consulta en el evento *BeforeOpen* del conjunto de datos no reportaba ventaja alguna con respecto a la preparación automática. Lo cual quiere decir que, al producirse este evento, la VCL ya ha preparado la consulta por sí misma y no agradece nuestra intervención.

Una solución de fuerza bruta para estos casos puede ser cerrar primeramente la consulta, prepararla y volver a abrirla, mediante un código parecido al siguiente:

```
void __fastcall TQueryForm::FormCreate(TObject *Sender)
{
    Query1->Close(); // Cerrar primeramente la consulta
    if (! Query1->Prepared)
        Query1->Prepare();
    Query1->Open(); // Reabrir la consulta
}
```

Pero conozco un truco mejor. Vaya a la sección **protected** de la declaración del módulo o formulario (si no hay, créela), y declare el siguiente método:

```
class TQueryForm : public TForm
{
    //...
protected:
    void __fastcall Loaded();
    // ...
};
```

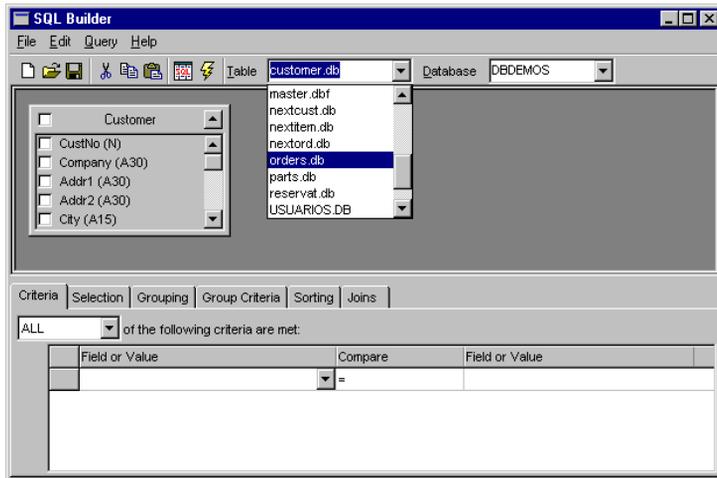
El método virtual *Loaded* es invocado por el formulario después de haber leído todos sus componentes hijos, pero antes de aplicar propiedades como *Active*, *Connected*, etc. Es decir, se llama justo en el momento que necesitamos. Aquí estamos sustituyendo su implementación predefinida mediante el siguiente método:

```
void __fastcall TQueryForm::Loaded()
{
    TForm::Loaded();
    Query1->Prepare();
}
```

Este truco de redefinir *Loaded* puede ser útil también para modificar durante la carga de una aplicación los parámetros de conexión de un componente *TDatabase*.

Visual Query Builder

Las versiones cliente/servidor de C++ Builder vienen acompañadas con una utilidad denominada *Visual Query Builder*, ó Constructor Visual de Consultas. No la busque dentro del directorio de ejecutables de C++ Builder, pues no es una herramienta que se ejecute por separado. Su objetivo es ayudar en la creación de consultas dentro de un componente *TQuery*, y se activa mediante la opción *SQL Builder* del menú de contexto asociado a un *TQuery*.



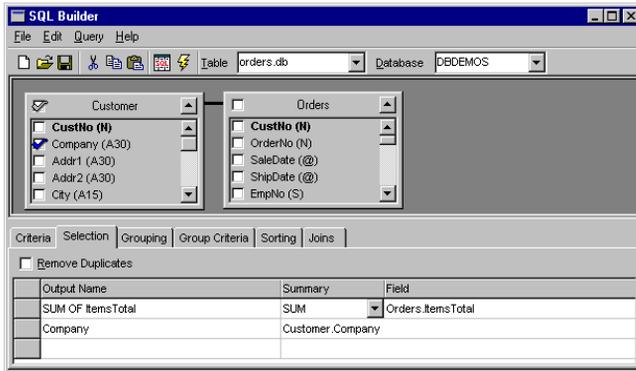
Para ilustrar el uso de la herramienta, diseñaremos una consulta que nos muestre un listado de clientes junto al total de dinero que han invertido en pedidos. Queremos además que el listado quede ordenado en forma descendente de acuerdo al total gastado; al que más nos haya comprado le enviaremos una tarjeta de felicitación por Navidades. Añadimos sobre un formulario vacío un componente *TQuery*, y asignamos *bdemos* a la propiedad *DatabaseName* de la consulta. Después pulsamos el botón derecho del ratón sobre la misma, y activamos el comando *SQL Builder*.

Para añadir tablas, debemos seleccionarlas en el combo de la izquierda. En nuestro caso, sabemos que los nombres de clientes se encuentran en la tabla *customer.db*, y que los totales por factura están en *orders.db*: las cabeceras de pedidos; añadimos entonces ambas tablas. No se preocupe si añade una tabla de más o de menos, pues más adelante se pueden eliminar y añadir tablas. Ya tenemos la cláusula **from** de la consulta.

Como el lector sabe, si en una consulta mencionamos dos tablas y no la relacionamos, obtendremos el temible *producto cartesiano*: todas las combinaciones de filas posibles, aún las más absurdas. Para relacionar estas dos tablas entre sí, arrastramos el campo *CustNo* de *customer* sobre el campo del mismo nombre de *orders*. Debe aparecer una línea que une a ambos campos. Si se ha equivocado, puede seleccionar la

línea, eliminarla con la tecla SUPR y reintentar la operación. Ya tenemos la cláusula **where**.

El próximo paso es indicar qué columnas se mostrarán como resultado de la consulta. Realice un doble clic sobre la columna *Company* de la tabla de clientes (el nombre de la empresa), y sobre *ItemsTotal*, de la tabla de pedidos (el total por factura). Esto determina la cláusula **select**. Si quiere ver el texto generado para la consulta, pulse un botón que dice “SQL”; si quiere ver los datos que devuelve la consulta, pulse el botón que tiene el rayo de Júpiter.



Todavía no está lista la consulta, pues se muestra una fila por cada pedido. Nos hace falta agrupar el resultado, para lo cual activamos la página *Selection*, y pulsamos el botón derecho del ratón sobre el campo *ItemsTotal*. En el menú que se despliega indicamos que este campo debe mostrar realmente una estadística (*Summary*). Entonces se divide en dos la celda, y en la nueva celda de la izquierda debemos seleccionar qué tipo de estadística necesitamos: en nuestro caso, *SUM*. Después ejecutamos la consulta generada, para comprobar el resultado de la operación; el texto de la consulta, hasta aquí, es el siguiente:

```

SELECT SUM( Orders.ItemsTotal ), Customer.Company
FROM "customer.db" Customer
INNER JOIN "orders.db" Orders
ON (Customer.CustNo = Orders.CustNo)
GROUP BY Customer.Company
    
```

Nos falta ordenar el resultado por la segunda columna; seleccionamos la página *Sorting*, añadimos la columna *ItemsTotal* al criterio de ordenación, y pedimos que se ordene descendentemente. La consulta final es la siguiente:

```
SELECT SUM( Orders.ItemsTotal ) Orders."SUM OF ItemsTotal",  
Customer.Company  
FROM "customer.db" Customer  
      INNER JOIN "orders.db" Orders  
      ON (Customer.CustNo = Orders.CustNo)  
GROUP BY Customer.Company  
ORDER BY Orders."SUM OF ItemsTotal" DESC
```

Por supuesto, para un programador experto en SQL puede ser más rápido y sencillo teclear directamente el texto de la instrucción. Recuerde también que ésta es una utilidad para tiempo de diseño. Si queremos permitir la generación visual de consultas al usuario final de nuestras aplicaciones, existen buenas herramientas en el mercado diseñadas para este propósito.

La interfaz de esta herramienta ha cambiado en C++ Builder 4 con respecto a versiones anteriores. El último paso efectuado, añadir un criterio de ordenación basado en una expresión, generaba en versiones anteriores una instrucción incorrecta que había que corregir a mano.

Comunicación cliente/servidor

TODA LA COMUNICACIÓN ENTRE EL Motor de Datos de Borland y los servidores SQL tiene lugar mediante sentencias SQL, incluso cuando el programador trabaja con tablas, en vez de con consultas. Para los desarrolladores en entornos cliente/servidor es de primordial importancia comprender cómo tiene lugar esta comunicación. En la mayoría de los casos, el BDE realiza su tarea eficientemente, pero hay ocasiones en las que tendremos que echarle una mano.

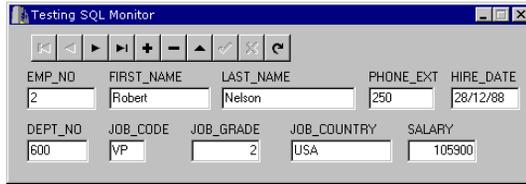
El propósito de este capítulo es enseñarle cómo detectar estas situaciones. Para lograrlo, veremos cómo el BDE traduce a instrucciones SQL las instrucciones de navegación y búsqueda sobre tablas y consultas. La forma en que se manejan las actualizaciones será estudiada más adelante.

Nuestra arma letal: SQL Monitor

Necesitamos un espía que nos cuente qué está pasando entre el servidor y nuestro cliente, y para esta labor contrataremos al SQL Monitor. Esta utilidad puede lanzarse desde el menú de programas de Windows, o directamente desde el menú *Database*, opción *SQL Monitor*, del propio C++ Builder. La ventana principal de este programa muestra las distintas instrucciones enviadas por el BDE y las respuestas que éste recibe del servidor. Podemos especificar qué tipos de instrucciones o de respuestas queremos mostrar en esta ventana mediante el comando de menú *Options | Trace options*. En ese mismo cuadro de diálogo se ajusta el tamaño del *buffer* que albergará las instrucciones. En principio, dejaremos las opciones tal y como vienen de la fábrica. Más adelante podrá desactivar algunas de ellas para mostrar una salida más compacta y legible.

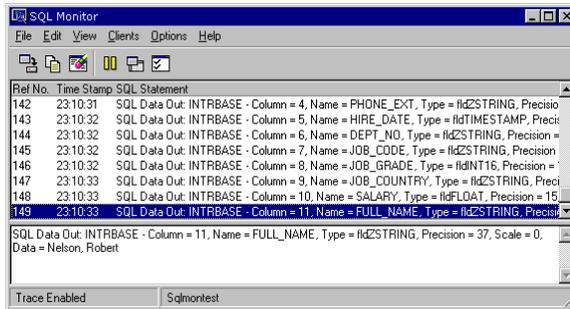
También necesitaremos una aplicación de prueba, que utilice una base de datos cliente/servidor. Para simplificar, utilizaremos un *TTable* asociado a la tabla *employee* del alias de InterBase *iblocal*. Crearemos los objetos de acceso a campos, los arrastraremos a la superficie del formulario, y añadiremos una barra de navegación. Importante: no se le ocurra utilizar una rejilla por el momento, pues se complicará la lectura e interpretación de los resultados de SQL Monitor.

La siguiente imagen muestra nuestra aplicación de pruebas en funcionamiento:



Apertura de tablas y consultas

Estando “apagada” la aplicación, lance SQL Monitor, mediante el método que más le guste. Luego, ejecute la aplicación y vea la salida generada:



¡Nada menos que 149 entradas¹⁷, solamente por abrir una tabla y leer el primer registro! Hagamos una prueba. Sustituya la tabla con una consulta que tenga la siguiente instrucción:

```
select * from employee
```

Repita ahora el experimento, ¡y verá que con sólo 15 entradas se puede comenzar a ejecutar la aplicación!

Hay un chiste (muy malo) acerca de unos científicos y un cangrejo. Al desdichado crustáceo le van arrancando las patas, mientras le ordenan verbalmente que se mueva. Al final, el animal se queda tieso en la mesa y no obedece las órdenes. Resultado anotado por los científicos: un cangrejo sin patas no oye. Parecida es la conclusión a la que llegaron algunos “gurús” de C++ Builder al ver estos resultados: hay que utilizar siempre consultas, en vez de tablas, si se está programando en un entorno cliente/servidor.

¹⁷ El número concreto de entradas puede variar, en función del sistema de bases de datos del servidor, de la ubicación del cliente, las opciones de trazado, etc. Los números que se den más adelante son también orientativos.

Y es que hay trampa en el asunto. ¿Se ha fijado que la consulta tiene la propiedad *RequestLive* igual a *False*? Cámbiela a *True* y repita la prueba, para que vea cómo vuelve a dispararse el contador de entradas en el monitor. Y pruebe después ir al último registro, tanto con la consulta como con la tabla, para que vea que la ventaja inicial de la consulta desaparece en este caso.

¿Qué está pasando? ¿Cómo podemos orientarnos entre la maraña de instrucciones del SQL Monitor? La primera regla de supervivencia es:

“Concéntrese en las instrucciones SQL Prepare y SQL Execute”

La razón es que éstas son las órdenes que envía el BDE al servidor. Repitiendo la apertura de la tabla, ¿con qué sentencias SQL nos tropezamos? Helas aquí:

```
select rdb$owner_name, rdb$relation_name, rdb$system_flag,
        rdb$view_blr, rdb$relation_id
from   rdb$relations
where  rdb$relation_name = 'employee'
```

El propósito de la sentencia anterior es comprobar si existe o no la tabla *employee*. Si esta instrucción diera como resultado un conjunto de filas vacío, fallaría la apertura de la tabla.

```
select r.rdb$field_name, f.rdb$field_type, f.rdb$field_sub_type,
        f.rdb$dimensions, f.rdb$field_length, f.rdb$field_scale,
        f.rdb$validation_blr, f.rdb$computed_blr,
        r.rdb$default_value, f.rdb$default_value, r.rdb$null_flag
from   rdb$relation_fields r, rdb$fields f
where  r.rdb$field_source = f.rdb$field_name and
        r.rdb$relation_name = 'employee'
order  by r.rdb$field_position asc
```

```
select i.rdb$index_name, i.rdb$unique_flag, i.rdb$index_type,
        f.rdb$field_name
from   rdb$indices i, rdb$index_segments f
where  i.rdb$relation_name = 'employee' and
        i.rdb$index_name = f.rdb$index_name
order  by i.rdb$index_id, f.rdb$field_position asc
```

```
select r.rdb$field_name, f.rdb$validation_blr, f.rdb$computed_blr,
        r.rdb$default_value, f.rdb$default_value, r.rdb$null_flag
from   rdb$relation_fields r, rdb$fields f
where  r.rdb$field_source = f.rdb$field_name and
        r.rdb$relation_name = 'employee'
order  by r.rdb$field_position asc
```

No hace falta ser un especialista en InterBase para darse cuenta de lo que está pasando. El BDE está extrayendo de las tablas del sistema la información sobre qué campos, índices y restricciones están definidas para esta tabla. Estos datos se almacenan dentro de las propiedades *FieldDefs* e *IndexDefs* de la tabla.

Finalmente, se abre la consulta básica para extraer datos de la tabla:

```
select emp_no, first_name, last_name, phone_ext, hire_date, dept_no,
       job_code, job_grade, job_country, salary, full_name
from   employee
order  by emp_no asc
```

Anote como detalle el que la consulta ordene las filas ascendentemente por el campo *Emp_No*, que es la clave primaria de esta tabla. Dentro de poco comprenderemos por qué.

La caché de esquemas

Traiga un botón al formulario y asocie el siguiente método con su evento *OnClick*:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    Table1->Close();
    Table1->Open();
}
```

Si pulsamos este botón durante la ejecución de la aplicación, veremos que la segunda vez que se abre la misma tabla no se produce la misma retahíla de sentencias que al inicio, sino que directamente se pasa a leer el primer registro del cursor. La explicación es que la tabla *ya* tiene la información de su esquema almacenada en las propiedades *FieldDefs* e *IndexDefs*. Esta es una buena optimización, porque disminuye el tráfico de datos en la red. Sin embargo, cada vez que se vuelve a ejecutar la aplicación partimos de cero, y hay que traer otra vez todos los datos del catálogo. Imagine una empresa con cincuenta empleados, todos conectando su ordenador a las 9:15 de la mañana (sí, porque de las 9:00 hasta entonces, café y cotilleo) y arrancando su aplicación ...

Este problema es el que resuelve la opción *ENABLE SCHEMA CACHE* del BDE, que vimos en el capítulo sobre la configuración del Motor de Datos. Así que ya sabe por qué es recomendable activar siempre la caché de esquemas.

Operaciones de navegación simple

Volvemos a las pruebas con la aplicación. Limpie el *buffer* de SQL Monitor, y pulse el botón de la barra de navegación de la aplicación que lo lleva al último registro de la tabla. Esta es la sentencia generada por el BDE, después de cerrar el cursor activo:

```
select emp_no, first_name, last_name, phone_ext, hire_date, dept_no,
       job_code, job_grade, job_country, salary, full_name
from   employee
```

```
order by emp_no desc
```

¡Ha cambiado el criterio de ordenación! Por supuesto, cuando el BDE ejecute la siguiente sentencia **fetch** el registro leído será el último de la tabla. Si seguimos navegando hacia atrás, con el botón *Prior*, el BDE solamente necesita ejecutar más instrucciones **fetch**, como en el *moon walk* de Michael Jackson: parece que nos movemos hacia atrás, pero en realidad nos movemos hacia delante.

Este truco ha sido posible gracias a la existencia de una clave primaria única sobre la tabla. Algunos sistemas SQL admiten que una clave primaria sea nula, permitiendo un solo valor nulo en esa columna, por supuesto. El problema es que, según el estándar SQL, al ordenar una secuencia de valores siendo algunos de ellos nulos, estos últimos siempre aparecerán en la misma posición: al principio o al final. Por lo que invertir el criterio de ordenación en el **select** no nos proporcionará los mismos datos en sentido inverso. El BDE no podrá practicar sus habilidades, y nos obligará a leer el millón de registros de la tabla a través de nuestra frágil y delicada red.

Precisamente eso es lo que sucede con un *TQuery*, sea actualizable o no. Cuando vamos al final del cursor, siempre se leen todos los registros intermedios. Cangrejo sin patas no oye, ¿cierto?

Quando se indica un criterio de ordenación para la tabla, ya sea mediante *IndexName* o *IndexFieldNames*, se cambia la cláusula **order by** del cursor del BDE. Sin embargo, sucede algo curioso cuando el criterio se especifica en *IndexName*: el BDE extrae los campos del índice para crear la sentencia SQL. Si la tabla es de InterBase, aunque el índice sea descendente la cláusula **order by** indicará el orden ascendente. Esto, evidentemente, es un *bug*, pues nos fuerza a utilizar una consulta si queremos ver las filas de una tabla ordenadas en forma descendente por determinada columna.

Búsquedas exactas con *Locate*

Añada al formulario un cuadro de edición (*TEdit*) y un botón. Con estos componentes vamos a organizar una búsqueda directa por código. Cree la siguiente respuesta al evento *OnClick* del botón:

```
void __fastcall TForm2::Button2Click(TObject *Sender)
{
    if (!Table1->Locate("EMP_NO", Edit1->Text, TLocateOptions())
        Beep());
}
```

Esta es una búsqueda exacta. Primero se prepara la siguiente instrucción:

```

select emp_no, first_name, last_name, phone_ext, hire_date, dept_no,
        job_code, job_grade, job_country, salary, full_name
from   employee
where  emp_no = ?

```

La sentencia tiene un parámetro. Para ejecutarla, el BDE utiliza antes una instrucción *Data in*, en la que proporciona el valor pasado en el segundo parámetro de *Locate* a la instrucción SQL. Lo interesante es lo que sucede cuando nos movemos con *Prior* y *Next* a partir del registro seleccionado. Si buscamos el registro anterior después de haber localizado uno, se genera la siguiente instrucción:

```

select emp_no, first_name, last_name, phone_ext, hire_date, dept_no,
        job_code, job_grade, job_country, salary, full_name
from   employee
where  emp_no < ?
order by emp_no desc

```

Es decir, el BDE abre un cursor descendente con los registros cuyo código es menor que el actual.

Búsquedas parciales

Cambiamos ahora la respuesta al evento *OnClick* del botón de búsqueda, para que efectúe una búsqueda parcial sobre la columna del apellido del empleado:

```

void __fastcall TForm2::Button2Click(TObject *Sender)
{
    if (! Table1->Locate("LAST_NAME", Edit1->Text,
        TLocateOptions() << loPartialKey))
        Beep();
}

```

Si tecleamos GUCK, encontraremos el registro de Mr. Guckenheimer, cuyo código de empleado es el 145. En este caso, el BDE dispara dos consultas consecutivas sobre la tabla. Esta es la primera:

```

select emp_no, first_name, last_name, phone_ext, hire_date, dept_no,
        job_code ,job_grade ,job_country ,salary ,full_name
from   employee
where  last_name = ?

```

La consulta anterior intenta averiguar si hay alguien cuyo apellido sea exactamente “Guck”. Si no sucede tal cosa, se dispara la segunda consulta:

```

select emp_no, first_name, last_name, phone_ext, hire_date, dept_no,
        job_code ,job_grade ,job_country ,salary ,full_name
from   employee
where  last_name > ?
order by last_name asc

```

Evidentemente, si un apellido comienza con “Guck”, es posterior alfabéticamente a este prefijo, y es esto lo que busca la segunda consulta: el primer registro cuyo apellido es mayor que el patrón de búsqueda tecleado. Eficiente, ¿no es cierto? Sin embargo, una pequeña modificación puede arruinar la buena reputación del BDE:

```
void __fastcall TForm2::Button2Click(TObject *Sender)
{
    if (! Table1->Locate("LAST_NAME", Edit1->Text,
        TLocateOptions() << loPartialKey << loCaseInsensitive))
        Beep();
}
```

La búsqueda ahora es también insensible a mayúsculas y minúsculas. Si teclaea nuevamente GUCK, estando al principio de la tabla, verá cómo en el SQL Monitor aparece una instrucción **fetch** por cada registro de la tabla. De lo que se puede deducir que si la tabla tuviera un millón de registros y estuviéramos buscando el último, podríamos dejar la máquina encendida e irnos a disfrutar de una semana de merecidas vacaciones.

El lector puede pensar, como pensó en su momento el autor, que la culpa de este comportamiento la tiene InterBase, que no permite índices insensibles a mayúsculas y minúsculas. No obstante, lo mismo sucede con Oracle y con SQL Server, como puede comprobar fácilmente quien tenga acceso a estos sistemas.

Una solución para búsquedas parciales rápidas

¿Qué pasa si para nuestra aplicación cliente/servidor son indispensables las búsquedas parciales rápidas insensibles a mayúsculas y minúsculas? Por ejemplo, queremos que el usuario pueda ir tecleando letras en un cuadro de edición, y en la medida en que teclee, se vaya desplazando la fila activa de la tabla que tiene frente a sí. En este caso, lo aconsejable es ayudar un poco al BDE.

Coloque un componente *TQuery* en el formulario que hemos estado utilizando, conéctelo a la base de datos y suminístrele la siguiente instrucción SQL:

```
select emp_no
from employee
where upper(last_name) starting with :patron
```

Con esta consulta pretendemos localizar los empleados cuyos apellidos comienzan con el prefijo que pasaremos en el parámetro *patron*. Edite la propiedad *Params* de la consulta y asígnele a este parámetro el tipo **string**. Como la consulta se ejecuta en el servidor, nos ahorramos todo el tráfico de red que implica el traernos todos los re-

gistros. Claro, ahora es responsabilidad del servidor el implementar eficientemente la consulta.

Por último, elimine el botón de búsqueda junto con el método asociado a su *OnClick*, y asocie esta respuesta al evento *OnChange* del cuadro de edición:

```
void __fastcall TForm1::Edit1Change(TObject *Sender)
{
    if (Edit1->Text == "")
        Table1->First();
    else
    {
        Query1->Params[0]->AsString = AnsiUpperCase(Edit1->Text);
        Query1->Open();
        try
        {
            if (Query1->Eof || ! Table1->Locate("EMP_NO",
                Query1->Fields->Fields[0]->Value,
                TLocateOptions()) Beep();
        }
        finally
        {
            Query1.Close;
        }
    }
}
```

Cuando el usuario teclea algo en el cuadro de edición, se ejecuta la consulta para recuperar los códigos de los posibles empleados. Evidentemente, si la consulta está vacía no existen tales empleados. En caso contrario, se escoge el primero de ellos y se busca su fila, utilizando una búsqueda exacta por el código mediante *Locate*, la cual ya sabemos que es rápida y segura.

El algoritmo anterior puede mejorarse con un par de trucos. Primero, puede prepararse la consulta antes de utilizarla, para ahorrarnos la compilación de la misma cada vez que busquemos algo. Y podemos hacer lo mismo que el BDE: utilizar una primera consulta que intente localizar el registro que corresponda exactamente al valor tecleado, antes de buscar una aproximación.

Búsquedas con filtros latentes

Para terminar este capítulo, analizaremos cómo el BDE implementa los filtros y rangos para las bases de datos cliente/servidor. Asigne la siguiente expresión en la propiedad *Filter* de la tabla:

```
Job_Grade = 5 and Salary >= 30000
```

A continuación active el filtro mediante la propiedad *Filtered*, y ejecute la aplicación. El cursor que abre el BDE añade la expresión de filtro dentro de su cláusula **where**:

```
select emp_no, first_name, last_name, phone_ext, hire_date, dept_no,
        job_code, job_grade, job_country, salary, full_name
from   employee
where  (job_grade = ? and salary >= ?)
order by emp_no asc
```

Aunque se están utilizando parámetros para las constantes, hasta aquí no hay ningún misterio. Llegamos a la conclusión, además, que los filtros se implementan eficientemente en sistemas cliente/servidor, pues la responsabilidad de la selección de filas queda a cargo del servidor. Sin embargo, es una conclusión prematura. Considere la inocente expresión:

```
Last_Name = 'B*'
```

Como hemos dicho en un capítulo anterior, con esta expresión se pretende seleccionar los empleados cuyos apellidos comienzan con la letra B. Al activar el filtro, podemos comprobar con SQL Monitor que el BDE trae al cliente todos los registros de la tabla, y los descarta localmente, a pesar de que la expresión anterior es equivalente a esta otra, que sabemos que se implementa eficientemente:

```
Last_Name >= 'B' and LastName < 'C'
```

El mismo problema se presenta cuando se añade la opción *foCaseInsensitive* a la propiedad *FilterOptions*. Por lo tanto, hay que tener cuidado con las expresiones que vamos a utilizar como filtro.

Los rangos se implementan de forma similar a las expresiones de filtro equivalentes. En tablas cliente/servidor es preferible, entonces, utilizar filtros en vez de rangos, pues son menos las limitaciones de los primeros: no hay que tener las filas ordenadas, se pueden establecer condiciones para los valores de varias filas simultáneamente...

Es interesante ver cómo se las arregla el BDE para navegar con *FindFirst*, *FindPrior*, *FindNext* y *FindLast* por las filas definidas por un filtro no activo, o filtro latente. Supongamos que *Filtered* sea *False*, y que a *Filter* le asignamos la expresión:

```
Job_Country <> 'USA'
```

Cuando se ejecuta el método *FindFirst*, el BDE lanza la siguiente sentencia:

```
select emp_no, first_name, last_name, phone_ext, hire_date, dept_no,  
        job_code, job_grade, job_country, salary, full_name  
from    employee  
where   job_country <> ?  
order  by emp_no asc
```

Para buscar el siguiente (*FindNext*), se utiliza el código de registro actual:

```
select emp_no, first_name, last_name, phone_ext, hire_date, dept_no,  
        job_code, job_grade, job_country, salary, full_name  
from    employee  
where   job_country <> ? and emp_no > ?  
order  by emp_no asc
```

Las operaciones *FindLast* y *FindPrior* se implementan de forma similar, invirtiendo solamente el orden de las filas. Y por supuesto, pueden surgir problemas si se utilizan filtros con búsquedas parciales o insensibles a mayúsculas y minúsculas.

He dejado el análisis de las operaciones de actualización sobre bases de datos SQL para los capítulos 26 y 27.

3

C++ Builder: actualizaciones y concurrencia

- Actualizaciones
- Actualizaciones mediante consultas
- Eventos de transición de estados
- Bases de datos y transacciones
- Sesiones
- Actualizaciones en caché

Parte

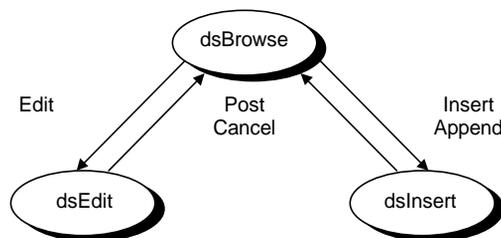
Actualizaciones

EL VIAJE MÁS LARGO comienza con un simple paso. Después de crear las tablas, hay que alimentarlas con datos. En este capítulo iniciaremos el estudio de los métodos de actualización. Se trata de un tema extenso, por lo cual nos limitaremos al principio a considerar actualizaciones sobre tablas aisladas, en entornos de un solo usuario, dejando para más adelante las actualizaciones coordinadas y el control de transacciones y concurrencia.

Los estados de edición y los métodos de transición

Como ya hemos explicado, cuando un conjunto de datos se encuentra en el estado *dsBrowse*, no es posible asignar valores a los campos, pues se produce una excepción. Para realizar modificaciones, hay que cambiar el estado del conjunto de datos a uno de los estados *dsEdit* ó *dsInsert*. Para realizar la transición a estos estados, debemos utilizar el método *Edit*, si es que deseamos realizar modificaciones sobre el registro activo; si queremos añadir registros debemos utilizar *Insert* ó *Append*.

```
// Para modificar                // Para añadir un nuevo registro
Table1->Edit();                 Table1->Append();
// ... secuencia de modificación ... // ... secuencia de inserción ...
```



Cuando estudiamos los controles de datos, vimos que se podía teclear directamente valores sobre los mismos, y modificar de este modo los campos, sin necesidad de llamar explícitamente al método *Edit*. Este comportamiento se controla desde la fuente de datos, el objeto *DataSource*, al cual se conectan los controles de datos. Si este componente tiene el valor *True* en su propiedad *AutoEdit*, una modificación en el

control provocará el paso del conjunto de datos al estado *Edit*. Puede ser deseable desactivar la edición automática cuando existe el riesgo de que el usuario modifique inadvertidamente los datos que está visualizando.

El método *Edit* debe releer siempre el registro actual, porque es posible que haya sido modificado desde otro puesto desde el momento en que lo leímos por primera vez.

Se puede aprovechar el comportamiento de *Edit*, que es válido tanto para bases de datos de escritorio como para bases de datos cliente/servidor, para releer el registro actual sin necesidad de llamar al método *Refresh*, que es potencialmente más costoso. Para asegurarse de que el contenido de la fila activa esté al día, ejecute el siguiente par de instrucciones:

```
Table1>Edit();
Table1>Cancel();
```

Es necesario llamar a *Cancel* para devolver la tabla al estado original *dsBrowse*.

Asignaciones a campos

Bien, ya tenemos los campos de la tabla en un modo receptivo a nuestras asignaciones. Ahora tenemos que realizar dichas asignaciones, y ya hemos visto cómo realizar asignaciones a un campo al estudiar la implementación de los campos calculados. Recordemos las posibilidades:

- Asignación por medio de variables de campos creadas mediante el Editor de Campos. Es la forma más eficiente de asignación, y la más segura, pues se comprueba su validez en tiempo de compilación.

```
tbClientes->Edit();
tbClientesLastInvoiceDate->Value = Date(); // Ultima factura
// ...
```

- Asignación mediante los objetos de campo creados en tiempo de ejecución, por medio de la función *FieldByName* o la propiedad *Fields*. *FieldByName* es menos estable frente a cambios de nombres de columnas, y respecto a errores tipográficos. Tampoco suministra información en tiempo de compilación acerca del tipo de campo. La propiedad *Fields* debe utilizarse solamente en casos especiales, pues nos hace depender además del orden de definición de las columnas.

```
tbClientes->Edit();
tbClientes->FieldByName("LastInvoiceDate")->AsDateTime = Date();
// ...
```

- Asignación mediante la propiedad *FieldValues*. Es la forma menos eficiente de todas, pues realiza la búsqueda de la columna y se asigna a una propiedad *Variant*, pero es quizás la más flexible.

```
tbClientes->Edit();
tbClientes->FieldValues["LastInvoiceDate"] = Date();
```

En Delphi tiene más atractivo utilizar *FieldValues*, pues ésta es la propiedad vectorial por omisión de la clase *TDataSet*. La siguiente instrucción sería correcta:

```
tbClientes['LastInvoiceDate'] := Date; // ¡Esto es Delphi!
```

Lamentablemente, C++ Builder no ofrece un equivalente para las propiedades por omisión, aunque hubiera sido fácil conseguirlo mediante la sobrecarga de operadores.

La asignación de valores a campos mediante la propiedad *FieldValues* nos permite asignar el valor **null** de SQL a un campo. Para esto, utilizamos la variante especial *Null*, definida en la unidad *System*:

```
tbClientes->FieldValues["LastInvoiceDate"] = Null;
```

Pero yo prefiero utilizar el método *Clear* del campo:

```
tbClientesLastInvoiceDate->Clear();
```

Es necesario tener bien claro que, aunque en Paradox y dBase un valor nulo se representa mediante una cadena vacía, esto no es así para tablas SQL.

Otra posibilidad es utilizar el método *Assign* para copiar el contenido de un campo en otro. Por ejemplo, si *Table1* y *Table2* son tablas con la misma estructura de campos, el siguiente bucle copia el contenido del registro activo de *Table2* en el registro activo de *Table1*; se asume que antes de este código, *Table1* se ha colocado en alguno de los estados de edición:

```
for (int i = 0; i < Table1->FieldCount; i++)
    Table1->Fields->Fields[i]->Assign(Table2->Fields->Fields[i]);
```

Cuando se copia directamente el contenido de un campo a otro con *Assign* deben coincidir los tipos de los campos y sus tamaños. Sin embargo, si los campos son campos BLOB, esta restricción se relaja. Incluso puede asignarse a estos campos el contenido de un memo o de una imagen:

```
Table1Foto->Assign(Imagel->Picture);
```

Por último, la propiedad *Modified* nos indica si se han realizado asignaciones sobre campos del registro activo que no hayan sido enviadas aún a la base de datos:

```
void GrabarOCancelar(TDataSet* ADataSet);
{
    if (ADataSet->State == dsEdit || ADataSet->State == dsInsert)
        if (ADataSet->Modified)
            ADataSet->Post();
        else
            ADataSet->Cancel();
}
```

El método *CheckBrowseMode*, de la clase *TDataSet*, es aproximadamente equivalente a nuestro *GrabarOCancelar*, como veremos más adelante.

Confirmando las actualizaciones

Una vez realizadas las asignaciones sobre los campos, podemos elegir entre confirmar los cambios o descartar las modificaciones, regresando en ambos casos al estado inicial, *dsBrowse*. Para confirmar los cambios se utiliza el método *Post*, indistintamente de si el conjunto de datos se encontraba en el estado *dsInsert* o en el *dsEdit*. El método *Post* corresponde, como ya hemos explicado, al botón que tiene la marca de verificación de las barras de navegación. Como también hemos dicho, *Post* es llamado implícitamente por los métodos que cambian la fila activa de un conjunto de datos, pero esta técnica es recomendada sólo para acciones inducidas por el usuario, nunca como recurso de programación. Siempre es preferible un *Post* explícito.

Si la tabla o consulta se encontraba inicialmente en el modo *dsInsert*, los valores actuales de los campos se utilizan para crear un nuevo registro en la tabla base. Si por el contrario, el estado inicial es *dsEdit*, los valores asignados modifican el registro activo. En ambos casos, y esto es importante, si la operación es exitosa la fila activa de la tabla corresponde al registro nuevo o al registro modificado.

Para salir de los estados de edición sin modificar el conjunto de datos, se utiliza el método *Cancel*, que corresponde al botón con la “X” en la barra de navegación. *Cancel* restaura el registro modificado, si el estado es *dsEdit*, o regresa al registro previo a la llamada a *Insert* ó *Append*, si el estado inicial es *dsInsert*. Una característica interesante de *Cancel* es que si la tabla se encuentra en un estado diferente a *dsInsert* ó *dsEdit* no pasa nada, pues se ignora la llamada.

Por el contrario, es una precondition de *Post* que el conjunto de datos se encuentre alguno de los estados de edición; de no ser así, se produce una excepción. Hay un método poco documentado, llamado *CheckBrowseMode*, que se encarga de asegurar que, tras su llamada, el conjunto de datos quede en el modo *dsBrowse*. Si la tabla o la consulta se encuentra en alguno de los modos de edición, se intenta una llamada a

Post. Si el conjunto de datos está inactivo, se lanza entonces una excepción. Esto nos ahorra repetir una y otra vez la siguiente instrucción:

```
if (Table1->State == dsEdit || Table1->State == dsInsert)
    Table1->Post();
```

Es muy importante, sobre todo cuando trabajamos con bases de datos locales, garantizar que una tabla siempre abandone el estado *Edit*. La razón, como veremos más adelante, es que para tablas locales *Edit* pide un bloqueo, que no es devuelto hasta que se llame a *Cancel* ó *Post*. Una secuencia correcta de edición por programa puede ser la siguiente:

```
Table1->Edit();
try
{
    // Asignaciones a campos ...
    Table1->Post();
}
catch(Exception&)
{
    Table1->Cancel();
    throw;
}
```

Diferencias entre *Insert* y *Append*

¿Por qué *Insert* y también *Append*? Cuando se trata de bases de datos SQL, los conceptos de inserción *in situ* y de inserción al final carecen de sentido, pues en este tipo de sistemas no existe el concepto de posición de registro. Por otra parte, el formato de tablas de dBase no permite una implementación eficiente de la inserción *in situ*, por lo cual la llamada al método *Insert* es siempre equivalente a una llamada a *Append*.

En realidad, en el único sistema en que estos dos métodos tienen un comportamiento diferente es en Paradox, en el caso especial de las tablas definidas sin índice primario. En este caso, *Insert* es realmente capaz de insertar el nuevo registro después del registro activo. Pero este tipo de tablas tiene poco uso, pues no se pueden definir índices secundarios en Paradox si no existe antes una clave primaria.

Pero la explicación anterior se refiere solamente al resultado final de ambas operaciones. Si estamos trabajando con una base de datos cliente/servidor, existe una pequeña diferencia entre *Insert* y *Append*, a tener en cuenta especialmente si paralelamente estamos visualizando los datos de la tabla en una rejilla. Cuando se realiza un *Append*, la fila activa se desplaza al final de la tabla, por lo que el BDE necesita leer los últimos registros de la misma. Luego, cuando se grabe el registro, la fila activa volverá a desplazarse, esta vez a la posición que le corresponde de acuerdo al criterio de ordenación activo. En el peor de los casos, esto significa releer dos veces la canti-

dad de registros que pueden aparecer simultáneamente en pantalla. Por el contrario, si se trata de *Insert*, solamente se produce el segundo desplazamiento, pues inicialmente la fila activa crea un “hueco” en la posición en que se encontraba antes de la inserción. Esta diferencia puede resultar o no significativa.

Si la tabla en que se está insertando contiene registros ordenados por algún campo secuencial, o por la fecha de inserción, y está ordenada por ese campo, es preferible utilizar *Append*, pues lo normal es que el registro quede definitivamente al final del cursor.

Como por azar...

¿Un pequeño ejemplo? Vamos a generar aleatoriamente filas para una tabla; esta operación es a veces útil para comprobar el funcionamiento de ciertas técnicas de C++ Builder. La tabla para la cual generaremos datos tendrá una estructura sencilla: una columna *Cadena*, de tipo cadena de caracteres, y una columna *Entero*, de tipo numérico. Supondremos que la clave primaria de esta tabla consiste en el campo *Cadena*; para el ejemplo actual es indiferente qué clave está definida. Lo primero será crear una función que nos devuelva una cadena alfabética aleatoria de longitud fija:

```

AnsiString RandomString(int Longitud)
{
    char* Vocales = "AEIOU";
    char LastChar = 'A';
    AnsiString Rslt;

    Rslt.SetLength(Longitud);
    for (int i = 1; i <= Longitud; i++)
    {
        LastChar = strchr("AEIOUNS", LastChar) ?
            random(26) + 'A' : Vocales[random(5)];
        Rslt[i] = LastChar;
    }
    return Rslt;
}

```

Me he tomado incluso la molestia de favorecer las secuencias consonante/vocal. El procedimiento que se encarga de llenar la tabla es el siguiente:

```

void LlenarTabla(TTable *Tabla, int CantRegistros)
{
    randomize();
    int Intentos = 3;
    while (CantRegistros > 0)
    try
    {
        Tabla->Append();
        Tabla->FieldValues["Cadena"] = RandomString(
            Tabla->FieldByName("Cadena")->Size);
    }
}

```

```

        Tabla->FieldValues["Entero"] = random(MAXINT);
        Tabla->Post();
        Intentos = 3;
        CantRegistros--;
    }
    catch(Exception&)
    {
        if (--Intentos == 0) throw;
    }
}

```

La mayoría de las excepciones se producirán por violaciones de la unicidad de la clave primaria. En definitiva, las excepciones son ignoradas, a no ser que sobrepasemos el número predefinido de intentos; esto nos asegura contra el desbordamiento de la capacidad de un disco y otros factores imprevisibles. El número de registros se decreta solamente cuando se produce una grabación exitosa.

NOTA IMPORTANTE

Cuando se trata de una base de datos SQL, el método de inserción masivo anterior es *mu*y ineficiente. En primer lugar: cada grabación (*Post*) abre y cierra una transacción, así que es conveniente agrupar varias grabaciones en una transacción. En segundo lugar, este algoritmo asume que estamos navegando sobre la tabla en la que insertamos, por lo que utiliza un *TTable*. Eso no es eficiente; más adelante veremos cómo utilizar un *TQuery* con parámetros para lograr más rapidez.

Métodos abreviados de inserción

Del mismo modo que *FindKey* y *FindNearest* son formas abreviadas para la búsqueda basada en índices, existen métodos para simplificar la inserción de registros en tablas y consultas. Estos son los métodos *InsertRecord* y *AppendRecord*:

```

void __fastcall TDataSet::InsertRecord(
    const System::TVarRec *Values, const int Values_Size);
void __fastcall TDataSet::AppendRecord(
    const System::TVarRec *Values, const int Values_Size);

```

En principio, por cada columna del conjunto de datos donde se realiza la inserción hay que suministrar un elemento en el vector de valores. El primer valor se asigna a la primera columna, y así sucesivamente. Pero también puede utilizarse como parámetro un vector con menos elementos que la cantidad de columnas de la tabla. En ese caso, las columnas que se quedan fueran se inicializan con el valor por omisión. El valor por omisión depende de la definición de la columna; si no se ha especificado otra cosa, se utiliza el valor nulo de SQL.

Si una tabla tiene tres columnas, y queremos insertar un registro tal que la primera y tercera columna tengan valores no nulos, mientras que la segunda columna sea nula, podemos pasar la constante *Null* en la posición correspondiente:

```
Table1->InsertRecord(ARRAYOFCONST(("Valor1", Null, "Valor3")));
// ...
Table1->AppendRecord(ARRAYOFCONST((
    RandomString(Table->FieldByName("Cadena")->Size),
    random(MAXINT))));
```

C++ Builder también ofrece el método *SetFields*, que asigna valores a los campos de una tabla a partir de un vector de valores:

```
Table1->Edit();
Table1->SetFields(ARRAYOFCONST(("Valor1", Null, "Valor3")));
Table1->Post();
```

El inconveniente de estos métodos abreviados se ve fácilmente: nos hacen dependientes del orden de definición de las columnas de la tabla. Se reestructura la tabla y ¡adiós inserciones!

El Gran Experimento

Hasta el momento hemos asumido que solamente nuestra aplicación tiene acceso, desde un solo puesto, a los datos con los que trabaja. Aún sin trabajar con bases de datos SQL en entornos cliente/servidor, esta suposición es irreal, pues casi cualquier escenario de trabajo actual cuenta con varios ordenadores conectados en una red puesto a puesto; a una aplicación para los formatos de datos más sencillos, como Paradox y dBase, se le exigirá que permita el acceso concurrente a éstos.

¿Qué sucede cuando varias aplicaciones intentan modificar simultáneamente el mismo registro? En vez de especular sobre la respuesta, lo más sensato es realizar un sencillo experimento que nos aclare el desarrollo de los acontecimientos. Aunque el experimento ideal debería involucrar al menos dos ordenadores, podremos arreglárnosla ejecutando dos veces la misma aplicación en la misma máquina. Lo más importante que descubriremos es que sucederán cosas diferentes cuando utilicemos bases de datos de escritorio y servidores SQL. La aplicación en sí será muy sencilla: una tabla, una fuente de datos (*TDataSource*), una rejilla de datos (*TDBGrid*) y una barra de navegación (*TDBNavigator*), esta última para facilitarnos las operaciones sobre la tabla.

El Gran Experimento: tablas locales

En su primera versión, la tabla debe referirse a una base de datos local, a una tabla en formato Paradox ó dBase. Para lograr que la misma aplicación se ejecute dos veces sin necesidad de utilizar el Explorador de Windows o el menú Inicio, cree el siguiente método en respuesta al evento *OnCreate* del formulario principal:

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    Top = 0;
    Width = Screen->Width / 2;
    Height = Screen->Height;
    if (CreateMutex(NULL, False,
        ExtractFileName(Application->ExeName).c_str()) != 0 &&
        GetLastError() == ERROR_ALREADY_EXISTS)
        Left = Screen->Width / 2;
    else
    {
        Left = 0;
        WinExec(Application->ExeName.c_str(), SW_NORMAL);
    }
}
```

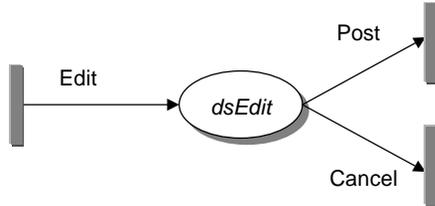
Para detectar la presencia de la aplicación he utilizado un semáforo binario soportado por Windows, llamado *mutex*, que se crea con la función *CreateMutex*. Para que se reconozca globalmente el semáforo hay que asignarle un nombre, que estamos formando a partir del nombre de la aplicación.

Ejecute dos instancias de la aplicación y efectúe entonces la siguiente secuencia de operaciones:

- Sitúese sobre un registro cualquiera de la rejilla, en la primera aplicación. Digamos, por conveniencia, que éste sea el primer registro.
- Ponga la tabla en estado de edición, pulsando el botón del triángulo (*Edit*) de la barra de navegación. El mismo efecto puede obtenerlo tecleando cualquier cosa en la rejilla, pues la fuente de datos tiene la propiedad *AutoEdit* con el valor por omisión, *True*. En cualquiera de estos dos casos, tenga cuidado de no cambiar la fila activa, pues se llamaría automáticamente a *Post*, volviendo la tabla al modo *dsBrowse*.
- Deje las cosas tal como están en la primera aplicación, y pase a la segunda.
- Sitúese en el mismo registro que escogió para la primera aplicación e intente poner la tabla en modo de edición, pulsando el correspondiente botón de la barra de navegación, o tecleando algo.

He escrito “intentar”, porque el resultado de esta acción es un mensaje de error: “Registro bloqueado por otro usuario”. Si tratamos de modificar una fila diferente no encontraremos problemas, lo que quiere decir que el bloqueo se aplica solamente al re-

gistro que se está editando en la otra aplicación, no a la tabla completa. También puede comprobar que el registro vuelve a estar disponible en cuanto guardamos las modificaciones realizadas en la primera aplicación, utilizando el botón de la marca de verificación (✓) o moviéndonos a otra fila. Lo mismo sucede si se cancela la operación.



La conclusión a extraer de este experimento es que, para las tablas locales, el método *Edit*, que se llama automáticamente al comenzar alguna modificación sobre la tabla, intenta colocar un bloqueo sobre la fila activa de la tabla. Este bloqueo puede eliminarse de dos formas: con el método *Post*, al confirmar los cambios, y con el método *Cancel*, al descartarlos.

El Gran Experimento: tablas SQL

Repetiremos ahora el experimento, pero cambiando el formato de la tabla sobre la cual trabajamos. Esta vez conecte con una tabla de InterBase, da lo mismo una que otra. Ejecute nuevamente la aplicación dos veces y siga estos pasos:

- En la primera aplicación, modifique el primer registro de la tabla, pero no confirme la grabación, dejando la tabla en modo de inserción.
- En la segunda aplicación, ponga la tabla en modo de edición y modifique el primer registro de la tabla. Esta vez no debe ocurrir error alguno. Grabe los cambios en el disco.
- Regrese a la primera aplicación e intente grabar los cambios efectuados en esta ventana.

En este momento, *sí* que tenemos un problema. La excepción se presenta con el siguiente mensaje: “No se pudo realizar la edición, porque otro usuario cambió el registro”. Hay que cancelar los cambios realizados, y entonces se releen los datos de la tabla, pues los valores introducidos por la segunda aplicación aparecen en la primera.

Pesimistas y optimistas

La explicación es, en este caso, más complicada. Acabamos de ver en acción un mecanismo *optimista* de control de edición. En contraste, a la técnica utilizada con las bases de datos locales se le denomina *pesimista*. Ya hemos visto que un sistema de control pesimista asume, al intentar editar un registro, que cualquier otro usuario puede estar editando este registro, por lo que para iniciar la edición “pide permiso” para hacerlo. Pedir permiso es el símil de colocar un bloqueo (*lock*): si hay realmente otro usuario editando esta fila se nos denegará dicho permiso.

C++ Builder transforma la negación del bloqueo en una excepción. Antes de lanzar la excepción, se nos avisa mediante el evento *OnEditError* de la tabla. En la respuesta a este evento tenemos la posibilidad de reintentar la operación, fallar con la excepción o fallar silenciosamente, con una excepción *EAbort*. En la lejana época en la que los Flintstones hacían de las suyas y la mayor parte de las aplicaciones funcionaban en modo *batch*, era de suma importancia decidir cuándo la aplicación que no obtenía un bloqueo se cansaba de pedirlo. Ahora, sencillamente, se le puede dejar la decisión al usuario. He aquí una simple respuesta al evento *OnEditError*, que puede ser compartido por todas las tablas locales de una aplicación:

```
void __fastcall TForm1::Table1EditError(TDataSet *DataSet,
    EDatabaseError *E, TDataAction &Action)
{
    if (MessageDlg(E->Message, mtWarning,
        TMsgDlgButton() << mbRetry << mbAbort, 0) == mrRetry)
        Action = daRetry;
    else
        Action = daAbort;
}
```

Otra posibilidad es programar un bucle infinito de reintentos. En este caso, es recomendable reintentar la operación transcurrido un intervalo de tiempo prudencial; cuando llamamos por teléfono y está la línea ocupada no marcamos frenéticamente el mismo número una y otra vez, sino que esperamos a que la otra persona termine su llamada. En este código muestro además cómo esperar un intervalo de tiempo aleatorio:

```
void __fastcall TForm1::Table1EditError(TDataSet *TDataSet,
    EDatabaseError *E, TDataAction &Action)
{
    // Esperar entre 1 y 2 segundos
    Sleep(1000 + random(1000));
    // Reintentar
    Action = daRetry;
}
```

El sistema pesimista es el más adecuado para bases de datos locales, pero para bases de datos SQL en entornos cliente/servidor, donde toda la comunicación transcurre a

través de la red, y en la que se trata de maximizar la cantidad de usuarios que pueden acceder a las bases de datos, no es la mejor política. En este tipo de sistemas, el método *Edit*, que marca el comienzo de la operación de modificación, no intenta colocar el bloqueo sobre la fila que va a cambiar. Es por eso que no se produce una excepción al poner la misma fila en modo de edición por dos aplicaciones simultáneas.

Las dos aplicaciones pueden realizar las asignaciones al *buffer* de registro sin ningún tipo de problemas. Recuerde que este *buffer* reside en el ordenador cliente. La primera de ellas en terminar puede enviar sin mayores dificultades la petición de actualización al servidor. Sin embargo, cuando la segunda intenta hacer lo mismo, descubre que el registro que había leído originalmente ha desaparecido, y en ese momento se aborta la operación mediante una excepción. Esta excepción pasa primero por el evento *OnPostError*, aunque en este caso lo mejor es releer el registro, si no se ha modificado la clave primaria, y volver a repetir la operación.

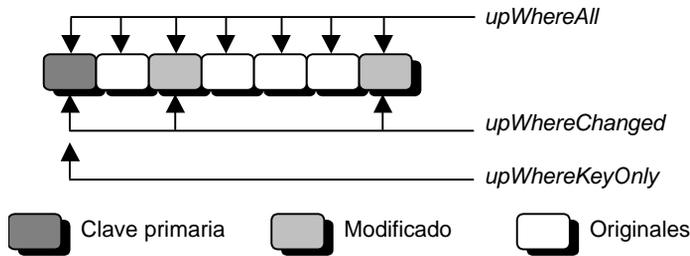
La suposición básica tras esta aparentemente absurda filosofía es que es poco probable que dos aplicaciones realmente traten de modificar el mismo registro a la vez. Piense, por ejemplo, en un cajero automático. ¿Qué posibilidad existe de que tratemos de sacar dinero al mismo tiempo que un buen samaritano aumenta nuestras existencias económicas? Siendo de breve duración este tipo de operaciones, ¿cuán probable es que la compañía de teléfonos y la de electricidad se topen de narices al saquearnos a principios de mes? Sin embargo, la razón de más peso para adoptar una filosofía optimista con respecto al control de concurrencia es que de esta manera disminuye el tiempo en que el bloqueo está activo sobre un registro, disminuyendo por consiguiente las restricciones de acceso sobre el mismo. Ahora este tiempo depende del rendimiento del sistema, no de la velocidad con que el usuario teclea sus datos después de la activación del modo de edición.

El modo de actualización

Cuando una tabla está utilizando el modo optimista de control de concurrencia, este mecanismo se configura de acuerdo a la propiedad *UpdateMode* de la tabla en cuestión. Esta propiedad nos dice qué algoritmo utilizarán C++ Builder y el BDE para localizar el registro original correspondiente al registro modificado. Los posibles valores son los siguientes:

Valor	Significado
<i>upWhereAll</i>	Todas las columnas se utilizan para buscar el registro a modificar.
<i>upWhereChanged</i>	Solamente se utilizan las columnas que pertenecen a la clave primaria, más las columnas modificadas.
<i>upWhereKeyOnly</i>	Solamente se utilizan las columnas de la clave primaria.

El valor por omisión es *upWhereAll*. Este valor es nuestro seguro de vida, pues es el más restrictivo de los tres. Es, en cambio, el menos eficiente, porque la petición de búsqueda del registro debe incluir más columnas y valores de columnas.



Aunque *upWhereKeyOnly* parezca una alternativa más atractiva, el emplear solamente las columnas de la clave puede llevarnos en el caso más general a situaciones en que dos aplicaciones entran en conflicto durante la modificación de un registro. Piense, por ejemplo, que estamos modificando el salario de un empleado; la clave primaria de la tabla de empleados es su código de empleado. Por lo tanto, si alguien está modificando en otro lugar alguna otra columna, como la fecha de contratación, las actualizaciones realizadas por nuestra aplicación pueden sobrescribir las actualizaciones realizadas por la otra aplicación. Si nuestra aplicación es la primera que escribe, tendremos un empleado con una fecha de contratación correcta y el salario sin corregir; si somos los últimos, el salario será el correcto (¡suerte que tiene el chico!), pero la fecha de contratación no estará al día.

Sin embargo, el valor *upWhereChanged* puede aplicarse cuando queremos permitir actualizaciones simultáneas en diferentes filas de un mismo registro, que no modifiquen la clave primaria; esta situación se conoce como *actualizaciones ortogonales*, y volveremos a mencionarlas en el capítulo sobre bases de datos remotas. Supongamos que nuestra aplicación aumenta el salario al empleado Ian Marteens. Si el valor de *UpdateMode* es *upWhereAll*, la instrucción SQL que lanza el BDE es la siguiente:

```
update Employee
set     Salary = 1000000      -- Me lo merezco
where  EmpNo = 666
       and FirstName = 'Ian'
       and LastName = 'Marteens'
       and Salary = 0        -- Este es el salario anterior
       and PhoneExt = '666'
       and HireDate = '1/1/97'
```

Por supuesto, si alguien cambia cualquier dato del empleado 666, digamos que la extensión telefónica, la instrucción anterior no encontrará registro alguno y se producirá un fallo de bloqueo optimista. En cambio, con *upWhereChanged* la instrucción generada sería:

```

update Employee
set Salary = 1000000 -- Me lo merezco
where EmpNo = 666
and Salary = 0 -- Este es el salario anterior

```

Lo único que se exige ahora es que no hayan cambiado a nuestras espaldas el código (nuestra identidad) o el salario (¡a ver si se pelean ahora por aumentarnos la paga!). Este tipo de configuración aumenta, por lo tanto, las posibilidades de acceso concurrente. Debe, sin embargo, utilizarse con cuidado, en particular cuando existen relaciones de dependencia entre las columnas de una tabla. El siguiente ejemplo es un caso extremo: se almacena en una misma fila la edad y la fecha de nacimiento; está claro que no tiene sentido permitir la modificación concurrente de ambas.

Pongamos ahora un ejemplo más real: una relación entre salario y antigüedad laboral, de manera que dentro de determinado rango de antigüedad solamente sean posibles los salarios dentro de cierto rango. Desde un puesto, alguien aumenta la antigüedad, pero desde otro puesto alguien disminuye el salario. Ambas operaciones darían resultados correctos por separado, pero al sumarse sus efectos, el resultado es inadmisibles. Sin embargo, la solución es sencilla: si existe la regla, debe haber algún mecanismo que se encargue de verificarla, ya sea una restricción **check** o un *trigger*. En tal caso, la primera de las dos modificaciones triunfaría sin problemas, mientras que la segunda sería rechazada por el servidor.

La relectura del registro actual

Hay otro detalle que no he mencionado con respecto a los fallos de bloqueos optimistas, y que podemos comprobar mediante SQL Monitor. Cuando se produce uno de estos fallos, el BDE relee sigilosamente el registro actual, a nuestras espaldas. Volvamos al ejemplo de la actualización de dos filas. Dos usuarios centran su atención simultáneamente en el empleado Marteens. El primero cambia su fecha de contrato al 1 de Enero del 96, mientras que el segundo aumenta su salario a un millón de dólares anuales. El segundo usuario es más rápido, y lanza primero el *Post*. Para complicar las cosas, supongamos que la tabla de empleados está configurada con la opción *upWhereAll*. A estas alturas debemos estar firmemente convencidos de que la actualización de la fecha de contrato va a fallar:

```

update Employee
set HireDate = '1/1/96' -- Cambiando la fecha
where EmpNo = 666
and FirstName = 'Ian'
and LastName = 'Marteens'
and Salary = 0 -- ¡Ya no, gracias a Dios!
and PhoneExt = '666'
and HireDate = '1/1/97'

```

Por supuesto, el usuario recibe una excepción. Sin embargo, observando la salida del SQL Monitor descubrimos que la aplicación lanza la siguiente instrucción:

```
select EmpNo, FirstName, LastName, Salary, PhoneExt, HireDate
from Employee
where EmpNo = 666 -- Sólo emplea la clave primaria
```

La aplicación ha releído nuestros datos, pero los valores actuales no aparecen en pantalla. ¿Para qué ha hecho esto, entonces? El primer usuario, el que insiste en corregir nuestra antigüedad, es un individuo terco. La fila activa de su tabla de empleados sigue en modo *dsEdit*, por lo cual puede repetir la operación *Post*. Esta es la nueva sentencia enviada:

```
update Employee
set HireDate = '1/1/96', -- Cambiando la fecha
Salary = 0 -- ¡No es justo!
where EmpNo = 666
and FirstName = 'Ian'
and LastName = 'Marteens'
and Salary = 1000000 -- Nos pilló
and PhoneExt = '666'
and HireDate = '1/1/97'
```

Es decir, al releer el registro, la aplicación está en condiciones de reintentar la grabación del registro que está en modo de edición. Además, se produce un efecto negativo: esta última grabación machaca cualquier cambio realizado concurrentemente desde otro puesto. Tenemos entonces un motivo adicional para utilizar *upWhere-Changed*, pues como es lógico, este tipo de situaciones no se producen en este modo de actualización.

Por desgracia, no tenemos propiedades que nos permitan conocer los valores actuales del registro durante el evento *OnPostError*. Si revisamos la documentación, encontraremos propiedades *OldValue*, *CurValue* y *NewValue* para cada campo. Sí, estas propiedades son precisamente las que nos hacen falta para diseñar manejadores de errores inteligentes ... pero solamente podemos utilizarlas si están activas las actualizaciones en caché, o si estamos trabajando con clientes Midas. Tendremos que esperar un poco más.

El método *Edit* sobre tablas cliente/servidor lanza una instrucción similar, para leer los valores actuales del registro activo. Puede suceder que ese registro que tenemos en nuestra pantalla haya sido eliminado por otro usuario, caso en el cual se produce el error correspondiente, que puede atraparse en el evento *OnEditError*. Este comportamiento de *Edit* puede aprovecharse para implementar un equivalente de *Refresh*, pero que solamente afecte al registro activo de un conjunto de datos:

```
Table1->Edit();
Table1->Cancel();
```

Es necesario llamar a *Cancel* para restaurar la tabla al modo *dsBrowse*.

Eliminando registros

La operación de actualización más sencilla sobre tablas y consultas es la eliminación de filas. Esta operación se realiza con un solo método, *Delete*, que actúa sobre la fila activa del conjunto de datos:

```
void TDataSet::Delete();
```

Después de eliminar una fila, se intenta dejar como fila activa la siguiente. Si ésta no existe, se intenta activar la anterior. Por ejemplo, para borrar todos los registros que satisfacen cierta condición necesitamos este código:

```
void __fastcall TForm1::BorrarTodosClick(TObject *Sender)
{
    Table1->First();
    while (! Table1->Eof)
        if (Condicion(Table1))
            Table1->Delete();
            // No se llama a Next() en este caso
        else
            Table1->Next();
}
```

Recuerde que eliminar una fila puede provocar la propagación de borrados, si existen restricciones de integridad referencial definidas de este modo.

Puede ser útil, en ocasiones, el método *EmptyTable*, que elimina todos los registros de una tabla. La tabla no tiene que estar abierta; si lo está, debe haberse abierto con la propiedad *Exclusive* igual a *True*.

dBase utiliza marcas lógicas cuando se elimina un registro. Paradox no las usa, pero deja un “hueco” en el sitio que ocupaba un registro borrado. Es necesaria una operación posterior para recuperar el espacio físico ocupado por estos registros. En el capítulo sobre programación con el BDE explicaremos cómo hacerlo.

Actualización directa vs variables en memoria

En la programación tradicional para bases de datos locales lo común, cuando se leen datos del teclado para altas o actualizaciones, es leer primeramente los datos en variables de memoria y, posteriormente, transferir los valores leídos a la tabla. Esto es lo habitual, por ejemplo, en la programación con Clipper. El equivalente en C++ Builder sería ejecutar un cuadro de diálogo “normal”, con controles *TEdit*, *TCombo*

Box y otros extraídos de la página *Standard* y, si el usuario acepta los datos tecleados, poner la tabla en modo de edición o inserción, según corresponda, asignar entonces el resultado de la edición a las variables de campo y terminar con un *Post*:

```

if (DialogoEdicion->ShowModal() == mrOk)
{
    Table1->Insert();
    Table1Nombre->Value = DialogoEdicion->Edit1->Text;
    Table1Edad->Value = StrToInt(DialogoEdicion->Edit2->Text);
    Table1->Post();
}

```

Esta técnica es relativamente simple, gracias a que la actualización tiene lugar sobre una sola tabla. Cuando los datos que se suministran tienen una estructura más compleja, es necesario recibir y transferir los datos utilizando estructuras de datos más complicadas. Por ejemplo, para entrar los datos correspondientes a una factura, hay que modificar varios registros que pertenecen a diferentes tablas: la cabecera del pedido, las distintas líneas de detalles, los cambios en el inventario, etc. Por lo tanto, los datos de la factura se leen en variables de memoria y, una vez que el usuario ha completado la ficha de entrada, se intenta su grabación en las tablas.

Por el contrario, el estilo preferido de programación para bases de datos en C++ Builder consiste en realizar siempre las actualizaciones directamente sobre las tablas, o los campos de la tabla, utilizando los componentes de bases de datos. ¿Qué ganamos con esta forma de trabajo?

- Evitamos el código que copia el contenido del cuadro de edición al campo. Esta tarea, repetida para cada pantalla de entrada de datos de la aplicación, puede convertirse en una carga tediosa, y el código generado puede consumir una buena parte del código total de la aplicación.
- La verificación de los datos es inmediata; para lograrlo en Clipper hay que duplicar el código de validación, o esperar a que la grabación tenga lugar para que salten los problemas. Ciertamente, el formato xBase es demasiado permisivo al respecto, y la mayor parte de las restricciones se verifican por la aplicación en vez de por el sistema de base de datos, pero hay que duplicar validaciones tales como la unicidad de las claves primarias. Esto también cuesta código y, lo peor, tiempo de ejecución.

Y ahora viene la pregunta del lector:

- Vale, tío listo, pero ¿qué haces si después de haber grabado la cabecera de un pedido y cinco líneas de detalles encuentras un error en la sexta línea? ¿Vas borrando una por una todas las inserciones e invirtiendo cada modificación hecha a registros existentes?

En este capítulo no podemos dar una solución completa al problema. La pista consiste, sin embargo, en utilizar el mecanismo de transacciones, que será estudiado más adelante, en combinación con las actualizaciones en caché. Así que, por el momento, confiad en mí.

Automatizando la entrada de datos

Lo que hemos visto, hasta el momento, es la secuencia de pasos necesaria para crear o modificar registros por medio de programación. Antes mencioné la posibilidad de utilizar componentes *data-aware* para evitar la duplicación de los datos tecleados por el usuario en variables y estructuras en memoria, dejando que los propios controles de edición actúen sobre los campos de las tablas. Supongamos que el usuario está explorando una tabla en una rejilla de datos. Si quiere modificar el registro que tiene seleccionado, puede pulsar un botón que le hemos preparado con el siguiente método de respuesta:

```
void __fastcall TForm1::bnEditarClick(TObject *Sender)
{
    Table1->Edit();
    DialogoEdicion->ShowModal();
    // Hasta aquí, por el momento ...
}
```

La ventana *DialogoEdicion* debe contener controles *data-aware* para modificar los valores de las columnas del registro activo de *Table1*. *DialogoEdicion* tiene también un par de botones para cerrar el cuadro de diálogo, los típicos *Aceptar* y *Cancelar*. He aquí lo que debe suceder al finalizar la ejecución del cuadro de diálogo:

- Si el usuario pulsa el botón *Aceptar*, debemos grabar, o intentar grabar, los datos introducidos.
- Si el usuario pulsa el botón *Cancelar*, debemos abandonar los cambios, llamando al método *Cancel*.

Lo más sencillo es verificar el valor de retorno de la función *ShowModal*, para decidir qué acción realizar:

```
// Versión inicial
void __fastcall TForm1::bnEditarClick(TObject *Sender)
{
    Table1->Edit();
    if (DialogoEdicion->ShowModal() == mrOk)
        Table1->Post();
    else
        Table1->Cancel();
}
```

Pero este código es muy malo. La llamada a *Post* puede fallar por las más diversas razones, y el usuario recibirá el mensaje de error con el cuadro de diálogo cerrado, cuando ya es demasiado tarde. Además, esta técnica necesita demasiadas instrucciones para cada llamada a un diálogo de entrada y modificación de datos.

Por lo tanto, debemos intentar la grabación cuando el cuadro de diálogo está todavía activo. El primer impulso del programador es asociar la pulsación del botón *Aceptar* con una llamada a *Post*, y una llamada a *Cancel* con el botón *Cancelar*. Pero esto nos obliga a escribir demasiado código cada vez que creamos una nueva ventana de entrada de datos, pues hay que definir tres manejadores de eventos: dos para los eventos *OnClick* de ambos botones, y uno para el evento *OnCloseQuery*, del formulario. En este último evento debemos preguntar si el usuario desea abandonar los cambios efectuados al registro, si es que existen y el usuario ha cancelado el cuadro de diálogo.

En mi opinión, el mejor momento para realizar la grabación o cancelación de una modificación o inserción es durante la respuesta al evento *OnCloseQuery*. Durante ese evento podemos, basándonos en el resultado de la ejecución modal que está almacenado en la propiedad *ModalResult* del formulario, decidir si grabamos, cancelamos o sencillamente si nos arrepentimos sinceramente de abandonar nuestros cambios. El código que dispara el evento *OnCloseQuery* tiene prevista la posibilidad de que se produzca una excepción durante su respuesta; en este caso, tampoco se cierra la ventana. Así, es posible evitar que se cierre el cuadro de diálogo si falla la llamada a *Post*. El algoritmo necesario se puede parametrizar y colocar en una función que puede ser llamada desde cualquier diálogo de entrada de datos:

```
bool PuedoCerrar(TForm *AForm, TDataSet *DS)
{
    if (AForm->ModalResult == mrOk)
        DS->Post();
    else if (! DS->Modified
        || (Application->MessageBox("¿Desea abandonar los cambios?",
            "Atención", MB_ICONQUESTION | MB_YESNO) == IDYES)
        DS->Cancel();
    else
        return False;
    return True;
}
```

La llamada típica a esta función es como sigue:

```
void __fastcall TDialogoEdicion::FormCloseQuery(TObject *Sender,
    bool &CanClose);
{
    CanClose = PuedoCerrar(this, Table1);
}
```

Esta función, *PuedoCerrar*, desempeñará un papel importante en este libro. A partir de este momento desarrollaremos variantes de la misma para aprovechar las diversas técnicas (transacciones, actualizaciones en caché) que vayamos estudiando.

La función *PuedoCerrar* puede definirse también como un método protegido en un formulario de prototipo. Este formulario puede utilizarse en el proyecto para que el resto de los diálogos de entrada de datos hereden de él, mediante la herencia visual.

Entrada de datos continua

Muchas aplicaciones están tan orientadas a la entrada de datos que, en aras de la facilidad de uso, no es conveniente tener que estar invocando una y otra vez la ficha de entrada de datos por cada registro a insertar. En este tipo de programas es preferible activar una sola vez el cuadro de diálogo para la entrada de datos, y redefinir el sentido del botón *Aceptar* de modo tal que grave los datos introducidos por el usuario y vuelva a preparar inmediatamente las condiciones para insertar un nuevo registro.

Es muy fácil modificar un cuadro de diálogo semejante a los desarrollados en la sección anterior, para que adopte el nuevo estilo de interacción. Basta con asignar *mrNone* a la propiedad *ModalResult* del botón *Aceptar*, de este modo, pulsar el botón no implica cerrar el cuadro de diálogo. En compensación, hay que teclear el siguiente código en la pulsación de dicho botón:

```
void __fastcall TForm1::bnOkClick(TObject *Sender)
{
    Table1->Post();
    Table1->Append();
}
```

Pero mi solución favorita, que nos permite utilizar menos eventos y teclear menos, es modificar el método *PuedoCerrar*, añadiéndole un parámetro que active o desactive la entrada de datos continua:

```
bool PuedoCerrar(TForm *AForm, TDataSet *DS, bool ModoContinuo)
{
    if (AForm->ModalResult == mrOk)
    {
        TDataSetState PrevState = DS->State;
        DS->Post();
        if (ModoContinuo && PrevState == dsInsert)
        {
            DS->Append();
            return False;
        }
    }
}
```

```

else if (! DS->Modified ||
    (Application->MessageBox("¿Desea abandonar los cambios?",
        "Atención", MB_ICONQUESTION | MB_YESNO) == IDYES)
    DS->Cancel();
else
    return False;
return True;
}

```

En este caso, la propiedad *ModalResult* del botón *Aceptar* debe seguir siendo *mrOk*. Lo único que varía es la forma de llamar a *PuedoCerrar* durante el evento *OnCloseQuery*:

```

void __fastcall TForm1::FormCloseQuery(TObject *Sender,
    bool &CanClose)
{
    CanClose = PuedoCerrar(this, Table1, True);
}

```


Actualizaciones mediante consultas

LA MAYORÍA DE LOS PROGRAMADORES NOVATOS PIENSA en SQL como en un lenguaje limitado a la selección de datos. Pero, como hemos visto, con SQL podemos también crear objetos en la base de datos, eliminarlos y modificar la información asociada a ellos. Cuando se utilizan instrucciones de ese tipo con un objeto *TQuery*, éste no puede tratarse como un conjunto de datos, que es lo que hemos visto hasta el momento.

Instrucciones del DML

Pongamos como ejemplo que queremos aumentarle el salario a todos los empleados de nuestra base de datos. Podemos entonces traer un objeto *TQuery* al formulario (o al módulo de datos), asignarle un valor a la propiedad *DatabaseName* y escribir la siguiente instrucción en la propiedad *SQL*:

```
update Employee
set Salary = Salary * (1 + :Puntos / 100)
```

Lo hemos complicado un poco utilizando un parámetro para el tanto por ciento del aumento. Ahora, nada de activar la consulta ni de traer fuentes de datos. Lo único que se necesita es ejecutar esta instrucción en respuesta a alguna acción del usuario:

```
void __fastcall TForm1::AumentoClick(TObject *Sender)
{
    Query1->ParamByName("Puntos")->AsInteger = 5;
    Query1->ExecSQL();
}
```

Como se ve, la ejecución de la consulta se logra llamando al método *ExecSQL*. Con anterioridad se ha asignado un valor al parámetro *Puntos*.

¿Cómo saber ahora cuántos empleados han visto aumentar sus ingresos? El componente *TQuery* tiene la propiedad *RowsAffected*, que puede ser consultada para obtener esta información:

```
void __fastcall TForm1::AumentoClick(TObject *Sender)
{
    Query1->ParamByName("Puntos")->AsInteger = 5;
    Query1->ExecSQL();
    ShowMessage(Format("Has traído la felicidad a %d personas",
        ARRAYOFCONST((Query1->RowsAffected))));
}
```

En el ejemplo previo, la instrucción estaba almacenada en un objeto *TQuery* incluido en tiempo de diseño. También se pueden utilizar instrucciones almacenadas en objetos creados en tiempo de ejecución. Se puede incluso programar un método genérico que nos ahorre toda la preparación y limpieza relacionada con la ejecución de una instrucción SQL. El siguiente procedimiento es uno de mis favoritos; en mi ordenador tengo una unidad en la cual incluyo ésta y otras funciones parecidas para utilizarlas de proyecto en proyecto:

```
int __fastcall EjecutarSql(const AnsiString ADatabase,
    const AnsiString Instruccion)
{
    std::auto_ptr<TQuery> query(new TQuery(NULL));
    query->DatabaseName = ADatabase;
    query->SQL->Text = Instruccion;
    query->ExecSQL();
    return query->RowsAffected;
}
```

El procedimiento se utilizará de este modo:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    EjecutarSql("dbdemos", "update employee "
        "set salary = salary * 1.05");
    // Note el espacio en blanco después de employee
    // y que no hay coma entre las dos últimas cadenas
}
```

He dividido la consulta en dos líneas simplemente por comodidad; en ANSI C++ dos constantes literales de cadena consecutivas se concatenan automáticamente durante la compilación. Recuerde además que *AnsiString* no tiene limitaciones de tamaño.

Almacenar el resultado de una consulta

Y ya que vamos de funciones auxiliares, he aquí otra de mis preferidas: una que guarda el resultado de una consulta en una tabla. Bueno, ¿y no existe acaso una ins-

trucción **insert...into** en SQL? Sí, pero con la función que vamos a desarrollar podremos mover el resultado de una consulta sobre *cierta* base de datos sobre *otra* base de datos, que podrá incluso tener diferente formato.

El procedimiento que explico a continuación se basa en el método *BatchMove* de los componentes *TTable*, que permite copiar datos de un conjunto de datos a una tabla. El conjunto de datos puede ser, por supuesto, una tabla, un procedimiento almacenado del tipo apropiado o, como en este caso, una consulta SQL.

```
void __fastcall GuardarConsulta(
    const AnsiString SourceDB, const AnsiString AQuery,
    const AnsiString TargetDB, const AnsiString ATableName)
{
    std::auto_ptr<TQuery> Q(new TQuery(NULL));
    Q->DatabaseName = SourceDB;
    Q->SQL->Text = AQuery;
    std::auto_ptr<TTable> T(new TTable(NULL));
    T->DatabaseName = TargetDB;
    T->TableName = ATableName;
    T->BatchMove(Q.get(), batCopy);
}
```

Una importante aplicación de este procedimiento puede ser la de realizar copias locales de datos procedentes de un servidor SQL en los ordenadores clientes.

¿Ejecutar o activar?

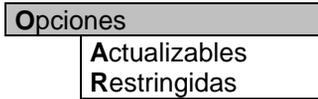
El ejemplo que voy a exponer en este epígrafe quizás no le sea muy útil para incluirlo en su próxima aplicación de bases de datos, pero le puede ayudar con la comprensión del funcionamiento de las excepciones. El ejercicio consiste en crear un intérprete para SQL, en el cual se pueda teclear cualquier instrucción y ejecutarla pulsando un botón. Y el adjetivo “cualquier” aplicado a “instrucción” es el que nos causa problemas: ¿cómo distinguir antes de ejecutar la instrucción si es una instrucción DDL, de manipulación o una consulta? Porque los dos primeros tipos necesitan una llamada al método *ExecSQL* para su ejecución, mientras que una consulta la activamos con el método *Open* o la propiedad *Active*, y necesitamos asociarle un *DataSource* para visualizar sus resultados.

Los componentes necesarios en este proyecto son los siguientes:

- *Memo1*: Un editor para teclear la instrucción SQL que deseamos ejecutar.
- *Query1*: Un componente de consultas, para ejecutar la instrucción tecleada.
- *DataSource1*: Fuente de datos, conectada al objeto anterior.
- *DBGrid1*: Rejilla de datos, conectada a la fuente de datos.

- *bnEjecutar*: Botón para desencadenar la ejecución de la instrucción.
- *cbAlias*: Un combo, con el estilo *csDropDownList*, para seleccionar un alias de los existentes.

Habilitaremos además un menú con opciones para controlar qué tipo de consultas queremos:



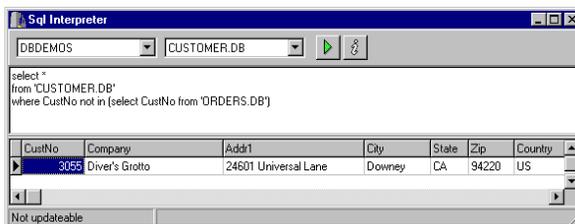
Estos dos comandos, a los cuales vamos a referirnos mediante las variable *miActualizables* y *miRestringidas*, funcionarán como casillas de verificación; la respuesta a ambas es el siguiente evento compartido:

```
void __fastcall TForm1::CambiarOpción(TObject *Sender)
{
    TMenuItem& mi = dynamic_cast<TMenuItem&>(*Sender);
    mi.Checked = ! mi.Checked;
}
```

Necesitaremos alguna forma de poder indicar el valor de la propiedad *DatabaseName* de la consulta. En vez de teclear el nombre de la base de datos, voy a utilizar un método de una clase que todavía no hemos estudiado: el método *GetDatabaseNames*, de la clase *TSession*. En el capítulo 30 se estudia esta clase con más detalle. Por el momento, sólo necesitamos saber que la función inicializa una lista de cadenas con los nombres de los alias disponibles en el momento actual. Esta inicialización tiene lugar durante la creación del formulario:

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    Session->GetDatabaseNames(cbAlias->Items);
    cbAlias->ItemIndex = 0;
}
```

Cuando pulsemos el botón *bnEjecutar*, debemos transferir el contenido del memo a la propiedad *SQL* de la consulta, y el nombre del alias seleccionado, que se encuentra en la propiedad *cbAlias.Text*, a la propiedad *DatabaseName* de la consulta. Una vez realizadas estas asignaciones, intentaremos abrir la consulta mediante el método *Open*.



¡Un momento! ¿No habíamos quedado en que había que analizar si el usuario había tecleado un **select**, o cualquier otra cosa? Bueno, esa sería una solución, pero voy a aprovechar un truco del BDE y la VCL: si intentamos activar una instrucción SQL que no es una consulta, se produce una excepción, ¡pero la instrucción se ejecuta de todos modos! Lo cual quiere decir que podemos situar la llamada a *Query1->Open* dentro de una instrucción **try/catch**, y capturar esa instrucción en particular, dejando que las restantes se propaguen para mostrar otros tipos de errores al usuario. La excepción que se produce en el caso que estamos analizando es de tipo *ENoResultSet*, y el mensaje de error, en la versión original, es “*Error creating cursor handle*”. He aquí la respuesta completa a la pulsación del botón *Ejecutar*.

```
void __fastcall TForm1::bnEjecutarClick(TObject *Sender)
{
    Query1->Close();
    Query1->DatabaseName = cbAlias->Text;
    Query1->SQL = Mem01->Lines;
    Query1->RequestLive = miActualizables->Checked;
    Query1->Constrained = miRestringidas->Checked;
    try
    {
        Query1->Open();
        if (Query1->CanModify)
            StatusBar1->SimpleText = "Consulta actualizable";
        else
            StatusBar1->SimpleText = "Consulta no actualizable";
    }
    catch(ENoResultSet& E)
    {
        StatusBar1->SimpleText = "Instrucción ejecutable";
    }
}
```

Adicionalmente, hemos colocado una barra de estado para mostrar información sobre el tipo de instrucción. Si es una instrucción ejecutable, se muestra el mensaje correspondiente. Si hemos tecleado una consulta, pedimos que ésta sea actualizable o no en dependencia del estado del comando de menú *miActualizable*; el resultado de nuestra petición se analiza después de la apertura exitosa del conjunto de datos, observando la propiedad *CanModify*.

El programa puede ampliarse fácilmente añadiendo soporte para comentarios. Por ejemplo, una forma rápida, aunque no completa, de permitir comentarios es sustituir la asignación a la propiedad SQL de la consulta por el siguiente código:

```
Query1->SQL->Clear();
for (int i = 0; i < Mem01->Lines->Count; i++)
    if (Mem01->Lines->Strings[i].TrimLeft().SubString(1, 2) != "--")
        Query1->SQL->Add(Mem01->Lines->Strings[i]);
```

De esta forma, se descartan todas las líneas cuyos dos primeros caracteres sean dos guiones consecutivos: el inicio de comentario del lenguaje modular de SQL. El lector

puede completar el ejemplo eliminando los comentarios que se coloquen al final de una línea.

Nuevamente como por azar....

¿Recuerda el ejemplo de inserción de registros aleatorios que vimos en el capítulo anterior? Vamos sustituir la inserción por medio del componente *TTable* por una más eficiente basada en *TQuery*. Lo primero que necesitamos es un componente de consultas, que asociamos a la base de datos correspondiente, y en cuya propiedad *SQL* introducimos la siguiente instrucción:

```
insert into TablaAleatoria(Entero, Cadena)
values (:Ent, :Cad)
```

El próximo paso es ir a la propiedad *Params* de la consulta, para identificar al parámetro *Ent* con el tipo *fiInteger*, y a *Cad* como *fiString*. El método que llena la tabla es el siguiente:

```
void LlenarTabla(TQuery *Query, int CantRegistros)
{
    randomize();
    Query->Prepare();
    try
    {
        int Intentos = 3;
        while (CantRegistros > 0)
            try
            {
                Query->ParamByName("ENT")->AsInteger = random(MAXINT);
                Query->ParamByName("CAD")->AsString = RandomString(35);
                Query->ExecSQL();
                Intentos = 3;
                CantRegistros--;
            }
            catch(Exception&)
            {
                if (--Intentos == 0) throw;
            }
        finally
        {
            Query->UnPrepare();
        }
    }
}
```

Una de las mayores ganancias de velocidad se obtiene al evitar preparar en cada paso la misma consulta. Por supuesto, y como dije al presentar antes este ejemplo, necesitaremos transacciones explícitas para lograr aún más rapidez.

Actualización sobre cursores de consultas

Una cosa es realizar una actualización ejecutando una instrucción SQL contenida en un *TQuery*, y otra bien distinta es utilizar una consulta **select** dentro de un *TQuery*, abrirla para navegación y realizar las modificaciones sobre este cursor. La primera técnica funciona de maravillas. La segunda es un desastre.

¿Por qué muchos programadores insisten, sin embargo, en la segunda técnica? Pues porque saben que una consulta no actualizable, contra un servidor SQL, tiene un tiempo de apertura infinitamente menor que el de una tabla, y tratan de extrapolar este resultado a las consultas actualizables, pensando que deben ser también más eficientes que las tablas. Falso: usted puede comprobar con el SQL Monitor que una consulta a la cual modificamos su *RequestLive* a *True* tarda lo mismo en iniciarse que una tabla, pues ejecuta el mismo protocolo de apertura.

Realmente existe una técnica para que las actualizaciones sobre cursores definidos por consultas sean más rápidas en general que las actualizaciones sobre tablas. Se trata de utilizar actualizaciones en caché, y emplear componentes *TUpdateSQL*, o interceptar el evento *OnUpdateRecord*, para evitar que el BDE decida el algoritmo de actualización. De esta forma se evitan el protocolo de apertura y otros problemas que mencionaré en esta sección. Pero debe tener en cuenta que tendrá que cambiar la metáfora de interacción del usuario.

De todos modos, el problema más grande que presentan las actualizaciones sobre consultas en estos momentos (BDE versión 5.01) es la inserción de un registro en un cursor. Me estoy arriesgando al poner sobre un papel con mi firma este tipo de afirmaciones: puede que un próximo parche del Motor de Datos o de la VCL resuelva definitivamente estos detalles, pero es preferible el riesgo a que usted pruebe una técnica, pensando que le va a ir bien, y cuando le salga mal se quede perplejo rumiando si la culpa es suya.

En concreto, cuando añadimos un registro a una consulta sobre la cual estamos navegando, desaparece siempre alguno de los registros anteriores. Si la consulta muestra inicialmente 10 registros, aunque creemos el oncenno seguirán mostrándose 10 registros en pantalla. ¿Solución? Refrescar el cursor...

... y aquí es surge la segunda dificultad con las consultas: no es posible refrescar una consulta abierta. Para colmo, el mensaje de error tiende a despistar: *"Table is not uniquely indexed"*. Ignore este mensaje: si necesita actualizar el cursor el único camino que tiene es cerrar y volver a abrir la consulta, y si es necesario, restaurar la posición del cursor. Como debe recordar, cualquier búsqueda sobre una consulta implica recuperar desde el servidor todos los registros intermedios (me gustaría que viese mi sonrisa sardónica mientras escribo esto).

No obstante, la actualización de registros existentes transcurre sin problemas. Las mismas consideraciones aplicables a las tablas (el modo de actualización, por ejemplo) son también válidas en esta situación.

Utilizando procedimientos almacenados

Para ejecutar un procedimiento almacenado desde una aplicación escrita en C++ Builder debemos utilizar el componente *TStoredProc*, de la página *Data Access* de la Paleta de Componentes. Esta clase hereda, al igual que *TTable* y *TQuery*, de la clase *TDBDataSet*. Por lo tanto, técnicamente es un conjunto de datos, y esto quiere decir que se le puede asociar un *TDataSource* para mostrar la información que contiene en controles de datos. Ahora bien, esto solamente puede hacerse en ciertos casos, en particular, para los procedimientos de selección de Sybase. ¿Recuerda el lector los procedimientos de selección de InterBase, que explicamos en el capítulo sobre *triggers* y procedimientos almacenados? Resulta que para utilizar estos procedimientos necesitamos una instrucción SQL, por lo cual la forma de utilizarlos desde C++ Builder es por medio de un componente *TQuery*.

En casi todos los casos, la secuencia de pasos para utilizar un *TStoredProc* es la siguiente:

- Asigne el nombre de la base de datos en la propiedad *DatabaseName*, y el nombre del procedimiento almacenado en *StoredProcName*.
- Edite la propiedad *Params*. *TStoredProc* puede asignar automáticamente los tipos a los parámetros, por lo que el objetivo de este paso es asignar opcionalmente valores iniciales a los parámetros de entrada.
- Si el procedimiento va a ejecutarse varias veces, prepare el procedimiento, de forma similar a lo que hacemos con las consultas paramétricas.
- Asigne, de ser necesario, valores a los parámetros de entrada utilizando la propiedad *Params* o la función *ParamByName*.
- Ejecute el procedimiento mediante el método *ExecProc*.
- Si el procedimiento tiene parámetros de salida, después de su ejecución pueden extraerse los valores de retorno desde la propiedad *Params* o por medio de la función *ParamByName*.

Pongamos por caso que queremos estadísticas acerca de cierto producto, del que conocemos su código. Necesitamos saber cuántos pedidos se han realizado, qué cantidad se ha vendido, por qué valor y el total de clientes interesados. Toda esa información puede obtenerse mediante el siguiente procedimiento almacenado:

```

create procedure EstadisticasProducto(CodProd int)
    returns (TotalPedidos int, CantidadTotal int,
            TotalVentas int, TotalClientes int)
as
declare variable Precio int;
begin
    select Precio
    from Articulos
    where Codigo = :CodProd
    into :Precio;
    select count(Numero), count(distinct RefCliente)
    from Pedidos
    where :CodProd in
        (select RefArticulo from Detalles
         where RefPedido = Numero)
    into :TotalPedidos, :TotalClientes;
    select sum(Cantidad),
           sum(Cantidad*:Precio*(100-Descuento)/100)
    from Detalles
    where Detalles.RefArticulo = :CodProd
    into :CantidadTotal, :TotalVentas;
end ^

```

Para llamar al procedimiento desde C++ Builder, configuramos en el módulo de datos un componente *TStoredProc* con los siguientes valores:

Propiedad	Valor
<i>DatabaseName</i>	El alias de la base de datos
<i>StoredProcName</i>	<i>EstadisticasProducto</i>

Después, para ejecutar el procedimiento y recibir la información de vuelta, utilizamos el siguiente código:

```

void __fastcall TForm1::MostrarInfo(TObject *Sender)
{
    AnsiString S;
    if (! InputQuery("Información", "Código del producto", S)
        || Trim(S) == "") return;
    TStoredProc *sp = modDatos->StoredProc1;
    sp->ParamByName("CodProd")->AsString = S;
    sp->ExecProc();
    ShowMessage(Format(
        "Pedidos: %d\nClientes: %d\nCantidad: %d\nTotal: %m",
        ARRAYOFCONST((
            sp->ParamByName("TotalPedidos")->AsInteger,
            sp->ParamByName("TotalClientes")->AsInteger,
            sp->ParamByName("CantidadTotal")->AsInteger,
            sp->ParamByName("TotalVentas")->AsFloat))));
}

```

Al total de ventas se le ha dado formato con la secuencia de caracteres *%m*, que traduce un valor real al formato nacional de moneda.

Eventos de transición de estados

LA PARTE VERDADERAMENTE COMPLICADA del diseño y programación con C++ Builder viene cuando se trata de establecer y forzar el cumplimiento de las reglas de consistencia, o como la moda actual las denomina en inglés: *business rules*. Algunas de estas reglas tienen un planteamiento sencillo, pues se tratan de condiciones que pueden verificarse analizando campos aislados. Ya sabemos cómo hacerlo, utilizando propiedades de los campos, como *Required*, *MinValue*, *MaxValue* y *EditMask* o, si la condición de validación es muy complicada, el evento *OnValidate*.

Otras reglas imponen condiciones sobre los valores de varios campos de un mismo registro a la vez; en este capítulo veremos cómo realizar este tipo de validaciones. Y las reglas más complejas requieren la coordinación entre los valores de varios registros de una o más tablas: claves primarias, integridad referencial, valores previamente calculados, etc. Para casi todas estas restricciones, necesitaremos el uso de los llamados *eventos de transición*, que se disparan automáticamente durante los cambios de estado de los conjuntos de datos. Aprenderemos a utilizar estos eventos tanto para imponer las condiciones correspondientes como para detectar el incumplimiento de las mismas.

Cuando el estado cambia...

Como corresponde a una arquitectura basada en componentes activos, la VCL ofrece una amplia gama de eventos que son disparados por los conjuntos de datos. Ya hemos visto algunos de estos eventos: *OnCalcFields*, para los campos calculados, *OnFilterRecord*, para el filtrado de filas. Pero la mayoría de los eventos producidos por las tablas y consultas se generan cuando se producen cambios de estado en estos componentes. Esta es la lista de los eventos disponibles en C++ Builder:

Método de transición	Eventos generados
<i>Open</i>	<i>BeforeOpen</i> , <i>AfterOpen</i>
<i>Close</i>	<i>BeforeClose</i> , <i>AfterClose</i>
<i>Edit</i>	<i>BeforeEdit</i> , <i>OnEditError</i> , <i>AfterEdit</i>
<i>Insert</i>	<i>BeforeInsert</i> , <i>OnNewRecord</i> , <i>AfterInsert</i>

Método de transición	Eventos generados
<i>Post</i>	<i>BeforePost, OnPostError, AfterPost</i>
<i>Cancel</i>	<i>BeforeCancel, AfterCancel</i>
<i>Delete</i>	<i>BeforeDelete, OnDeleteError, AfterDelete</i>

Además de los eventos listados, falta mencionar a los eventos *OnUpdateError* y *OnUpdateRecord*, relacionados con las actualizaciones en caché. Existen también los eventos *BeforeScroll* y *AfterScroll*, asociados a los cambios de fila activa, no a cambios de estado.

Los eventos *BeforeScroll* y *AfterScroll* son más específicos que el evento *OnChange* del componente *TDataSource*. Este último evento, además de dispararse cuando cambia la fila activa, también se genera cuando se realizan cambios en los valores de los campos de la fila activa.

Algunos de los eventos de la lista están relacionados con lo que sucede antes y después de que la tabla ejecute el método de transición; sus nombres comienzan con los prefijos *Before* y *After*. Otros están relacionados con los errores provocados durante este proceso: *OnEditError*, *OnDeleteError* y *OnPostError*. Queda además *OnNewRecord*, que es disparado por la tabla después de entrar en el estado *dsInsert*, y a continuación del cual se limpia el indicador de modificaciones del registro activo. Es natural que, ante tanta variedad de eventos, el programador se sienta abrumado y no sepa por dónde empezar. No obstante, es relativamente sencillo marcarnos una guía acerca de qué eventos tenemos que interceptar de acuerdo a nuestras necesidades.

Reglas de empresa: ¿en el servidor o en el cliente?

Muchas veces, los programadores que conocen algún sistema SQL y que están aprendiendo C++ Builder se plantean la utilidad de los eventos de transición de estado, si el sistema al que se accede es precisamente un sistema SQL cliente/servidor. El argumento es que la implementación de las reglas de empresa debe tener lugar preferentemente en el servidor, por medio de *triggers* y procedimientos almacenados. La implementación de reglas en el servidor nos permite ahorrar código en los clientes, nos ofrece más seguridad y, en ocasiones, más eficiencia.

Sin embargo, el hecho básico del cual debemos percatarnos es el siguiente: los eventos de transición de C++ Builder se refieren a *acciones que transcurren durante la edición*, en el lado cliente de la aplicación. Aunque en algunos casos realmente tenemos la libertad de implementar alguna regla utilizando cualquiera de estas dos técnicas, hay ocasiones en que esto no es posible. La siguiente tabla muestra una equivalencia entre los *triggers* y los eventos de transición de estados:

Trigger	Insert	Update	Delete
Antes	<i>BeforePost</i>	<i>BeforePost</i>	<i>BeforeDelete</i>
Después	<i>AfterPost</i>	<i>AfterPost</i>	<i>AfterDelete</i>

Como se aprecia, solamente un pequeño grupo de eventos de transición están representados en esta tabla. No hay equivalente en SQL a los eventos *BeforeEdit*, *AfterCancel* ó *BeforeClose*, por mencionar algunos. Por otra parte, incluso cuando hay equivalencia en el comportamiento, desde el punto de vista del usuario puede haber diferencias. Por ejemplo, como veremos en la siguiente sección, el evento *OnNewRecord* puede utilizarse para asignar valores por omisión a los campos de registros nuevos. Esto mismo puede efectuarse con el *trigger before insert*. Sin embargo, si utilizamos *OnNewRecord*, el usuario puede ver durante la edición del registro los valores que van a tomar las columnas implicadas, cosa imposible si se implementa el *trigger*.

Por lo tanto, lo ideal es una mezcla de ambas técnicas, que debe decidirse en base a los requisitos de cada aplicación. Más adelante, cuando estudiemos las actualizaciones en caché, profundizaremos más en este tema.

Inicialización de registros: el evento *OnNewRecord*

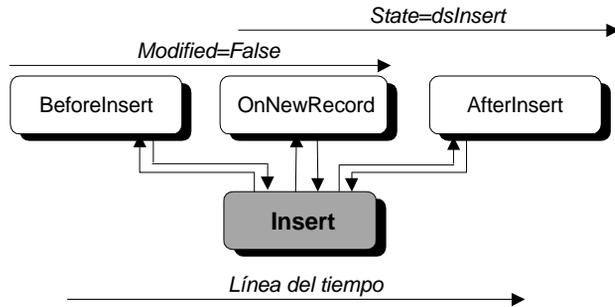
Posiblemente el evento más popular de los conjuntos de datos sea *OnNewRecord*. Aunque su nombre no comienza con *After* ni con *Before*, se produce durante la transición al estado *dsInsert*, cuando ya el conjunto de datos se encuentra en dicho estado, pero antes de disparar *AfterInsert*. Este evento se utiliza para la asignación de valores por omisión durante la inserción de registros.

La diferencia entre *OnNewRecord* y *AfterInsert* consiste en que las asignaciones realizadas a los campos durante el primer evento *no* marcan el registro actual como modificado. ¿Y en qué notamos esta diferencia? Tomemos por ejemplo, la tabla *employee.db* de la base de datos *bcdemos*, y visualicémosla en una rejilla. Interceptemos el evento *OnNewRecord* y asociémosle el siguiente procedimiento:

```
void __fastcall TForm1::Table1NewRecord(TDataSet *DataSet)
{
    DataSet->FieldValues["HireDate"] = Date();
}
```

Estamos asignando a la fecha de contratación la fecha actual, algo lógico en un proceso de altas. Ahora ejecute el programa y realice las siguientes operaciones: vaya a la última fila de la tabla y utilizando la tecla de cursor FLECHA ABAJO cree una nueva fila moviéndose más allá de este último registro. Debe aparecer un registro vacío, con la salvedad de que el campo que contiene la fecha de contratación ya tiene asignado un valor. Ahora dé marcha atrás sin tocar nada y el nuevo registro desaparecerá, precisamente porque, desde el punto de vista de C++ Builder, no hemos modificado nada

en el mismo. Recuerde que esta información nos la ofrece la propiedad *Modified* de los conjuntos de datos.



Para notar la diferencia, desacople este manejador de eventos de *OnNewRecord* y asíócielo al evento *AfterInsert*. Si realiza estas mismas operaciones en la aplicación resultante verá cómo si se equivoca y crea un registro por descuido, no puede deshacer la operación con sólo volver a la fila anterior.

En el capítulo 18 vimos que la propiedad *DefaultExpression* de los campos sirve para asignar automáticamente valores por omisión sencillos a estos componentes. Sin embargo, dadas las limitaciones de las expresiones SQL que son reconocidas por el intérprete local, en muchos casos hay que recurrir también al evento *OnNewRecord* para poder completar la inicialización. Al parecer, existe un *bug* que impide que estos dos mecanismos puedan utilizarse simultáneamente. Por lo tanto, si necesita utilizar *OnNewRecord* en una tabla, es muy importante que limpie todas las propiedades *DefaultExpression* de los campos de dicha tabla. Por lo menos, hasta nuevo aviso.

Validaciones a nivel de registros

En los casos en que se necesita imponer una condición sobre varios campos de un mismo registro, la mejor forma de lograrlo es mediante el evento *BeforePost*. Se verifica la condición y, en el caso en que no se cumpla, se aborta la operación con una excepción, del mismo modo que hacíamos con el evento *OnValidate* de los campos. Hay que tener en cuenta que *BeforePost* se llama igualmente para las inserciones y las modificaciones. Para distinguir entre estas dos situaciones, tenemos que utilizar, por supuesto, la propiedad *State* del conjunto de datos. Por ejemplo, para impedir que un empleado nuevo gane una cantidad superior a cierto límite salarial podemos programar la siguiente respuesta a *BeforePost*:

```

void __fastcall TmodDatos::tbEmpleadosBeforePost(TDataSet *DataSet)
{
    if (tbEmpleados->State == dsInsert)
        if (tbEmpleadosSalary->Value > TopeSalarial)
            DatabaseError("Demasiado dinero para un aprendiz", 0);
}

```

Asumimos que *TopeSalarial* es una constante definida en el módulo, o una variable convenientemente inicializada. El procedimiento *DatabaseError* ya ha sido utilizado en el capítulo sobre acceso a campos, durante la respuesta al evento *OnValidate*.

Este manejador de eventos no impedirá, sin embargo, modificar posteriormente el salario una vez realizada la inserción. Además, hubiera podido implementarse en respuesta al evento *OnValidate* del componente *tbEmpleadosSalary*, pues únicamente verifica los valores almacenados en un solo campo. Una alternativa válida tanto para las altas como para las modificaciones es realizar la validación comprobando la antigüedad del trabajador, fijando un tope para los empleados que llevan menos de un año en la empresa:

```

void __fastcall TmodDatos::tbEmpleadosBeforePost(TDataSet *DataSet)
{
    if (Date() - tbEmpleadosHireDate->Value < 365
        && tbEmpleadosSalary->Value > TopeSalarial)
        DatabaseError("¡Este es un enchufado!", 0);
}

```

Este criterio de validación no puede implementarse correctamente utilizando sólo los eventos *OnValidate*, pues involucra simultáneamente a más de un campo de la tabla. Observe el uso de la función *Date* (fecha actual), y de la resta entre fechas para calcular la diferencia en días.

Antes y después de una modificación

Cuando simulamos *triggers* en C++ Builder tropezamos con un inconveniente: no tenemos nada parecido a las variables de contexto *new* y *old*, para obtener los valores nuevos y antiguos del registro que se actualiza. Si necesitamos estos valores debemos almacenarlos manualmente, y para ello podemos utilizar el evento *BeforeEdit*. Luego, podemos aprovechar los valores guardados anteriormente en el evento *AfterPost*, para efectuar acciones similares a las de un *trigger*.

Pongamos por caso que queremos mantener actualizada la columna *ItemsTotal* de la tabla de pedidos cada vez que se produzca un cambio en las líneas de detalles asociadas. Para simplificar nuestra exposición, supondremos que sobre la tabla de detalles se ha definido un campo calculado *SubTotal*, que se calcula multiplicando la cantidad de artículos por el precio extraído de la tabla de artículos y que tiene en cuenta el descuento aplicado. Tenemos que detectar las variaciones en este campo para refle-

jarlas en la columna *ItemsTotal* de la fila activa de pedidos. Asumiremos, por supuesto, que existe una relación *master/detail* entre las tablas de pedidos y detalles.

Como primer paso, necesitamos declarar una variable, *OldSubTotal*, para almacenar el valor de la columna *SubTotal* antes de cada modificación. Esto lo hacemos en la declaración de la clase correspondiente al módulo de datos, en su sección **private** (pues esta declaración no le interesa a nadie más):

```
class TmodDatos : public TDataModule
{
    // ...
private:
    Currency OldSubTotal;
};
```

Cuando declaramos una variable, debemos preocuparnos inmediatamente por su inicialización. Esta variable, en particular, por estar definida dentro de una clase basada en la VCL, se inicializa con 0 al crearse el módulo de datos. Cuando vamos a realizar modificaciones en una línea de detalles, necesitamos el valor anterior de la columna *SubTotal*:

```
void __fastcall TmodDatos::tbLineasBeforeEdit(TDataSet *DataSet)
{
    OldSubTotal = tbLineas->FieldValues["SubTotal"];
}
```

Como la actualización de *ItemsTotal* se realizará en el evento *AfterPost*, que también se dispara durante las inserciones, es conveniente inicializar también *OldSubTotal* en el evento *BeforeInsert*:

```
void __fastcall TmodDatos::tbLineasBeforeInsert(TDataSet *DataSet)
{
    OldSubTotal = 0;
}
```

El mismo resultado se obtiene si la inicialización de *OldSubTotal* para las inserciones se realiza en el evento *OnNewRecord*, pero este evento posiblemente tenga más código asociado, y es preferible separar la inicialización de variables de contexto de la asignación de valores por omisión.

Entonces definimos la respuesta al evento *AfterPost* de la tabla de líneas de detalles:

```
void __fastcall TmodDatos::tbLineasAfterPost(TDataSet *DataSet)
{
    Currency DiffSubTotal =
        tbLineas->FieldValues["SubTotal"] - OldSubTotal;
    if (DiffSubTotal != 0)
    {
        if (tbPedidos->State!=dsEdit && tbPedidos->State!=dsInsert)
            tbPedidos->Edit();
    }
}
```

```

        tbPedidos->FieldValues["ItemsTotal"] =
            tbPedidos->FieldValues["ItemsTotal"] + DiffSubTotal;
    }
}

```

Como la tabla de pedidos es la tabla maestra de la de detalles, no hay necesidad de localizar el registro del pedido, pues es el registro activo. He decidido no llamar al método *Post* sobre la tabla de pedidos, asumiendo que estamos realizando la edición simultánea de la tabla de pedidos con la de líneas de detalles. En los capítulos sobre transacciones y actualizaciones en caché veremos métodos para automatizar la grabación de datos durante la edición e inserción de objetos complejos.

Le propongo al lector que implemente el código necesario para mantener actualizado *ItemsTotal* durante la eliminación de líneas de detalles.

Propagación de cambios en cascada

Una aplicación útil de los eventos de transición de estados es la propagación en cascada de cambios cuando existen restricciones de integridad referencial entre tablas. Normalmente, los sistemas SQL y Paradox permiten especificar este comportamiento en forma declarativa durante la definición de la base de datos; ya hemos visto cómo hacerlo al estudiar las restricciones de integridad referencial. Pero hay sistemas que no implementan este recurso, o lo implementan parcialmente. En tales casos, tenemos que recurrir a *triggers* o a eventos de transición de estados.

Si estamos programando para tablas dBase anteriores a la versión 7, y queremos establecer la propagación de borrados desde la tabla de pedidos a la de líneas de detalles, podemos crear un manejador de eventos en este estilo:

```

void __fastcall TmodDatos::tbPedidosBeforeDelete(TDataSet *DataSet)
{
    tbLineas->First();
    while (! tbLineas->Eof)
        tbLineas->Delete();
}

```

Observe que no se utiliza el método *Next* para avanzar a la próxima fila de la tabla dependiente, pues el método *Delete* se encarga de esto automáticamente. Curiosamente, C++ Builder trae un ejemplo de borrado en cascada, pero el ciclo de borrado se implementa de este modo:

```

while (Tabla->RecordCount > 0)
    // ... etcétera ...

```

Este código funciona, por supuesto, pero si la tabla pertenece a una base de datos SQL, *RecordCount* es una propiedad peligrosa, como ya hemos explicado.

Si lo que queremos, por el contrario, es prohibir el borrado de pedidos con líneas asociadas, necesitamos esta otra respuesta:

```
void __fastcall TmodDatos::tbPedidosBeforeDelete(TDataSet *DataSet)
{
    if (! tbLineas->IsEmpty())
        DatabaseError("Este pedido tiene líneas asociadas", 0);
}
```

Incluso si el sistema permite la implementación de los borrados en cascada, puede interesarnos interceptar este evento, para dejar que el usuario decida qué hacer en cada caso. Por ejemplo:

```
void __fastcall TmodDatos::tbPedidosBeforeDelete(TDataSet *DataSet)
{
    tbLineas->First();
    if (! tbLineas->Eof)
        if (MessageDlg("¿Eliminar líneas de detalles?",
            mtConfirmation, TMsgDlgButtons()<<mbYes<<mbNo, 0)=mrYes)
            do
            {
                tbLineas->Delete();
            }
            while (! tbLineas->Eof);
    else
        Abort();
}
```

Note el uso de *Abort*, la excepción silenciosa, para interrumpir la ejecución de la operación activa y evitar la duplicación innecesaria de mensajes al usuario.

Actualizaciones coordinadas master/detail

He aquí otro caso especial de coordinación mediante eventos de conjuntos de datos, que es aplicable cuando existen pares de tablas en relación *master/detail*. Para precisar el ejemplo, supongamos que son las tablas *orders.db* e *items.db*, del alias *bcemos*: una tabla de pedidos y su tabla asociada de líneas de detalles. En la mayoría de las ocasiones existe una restricción de integridad referencial definida en la tabla esclava. Por lo tanto, para poder grabar una fila de la tabla de detalles tiene que existir previamente la correspondiente fila de la tabla maestra. Para garantizar la existencia de la fila maestra, podemos dar respuesta al evento *BeforeInsert* de la tabla dependiente:

```
void __fastcall TmodDatos::tbDetallesBeforeInsert(TDataSet *DataSet)
{
    if (tbPedidos->State == dsInsert)
    {
        tbPedidos->Post();
        tbPedidos->Edit();
    }
}
```

En este caso, además, hemos colocado automáticamente la tabla maestra en modo de edición, por si son necesarias más actualizaciones sobre la cabecera, como el mantenimiento de la columna *ItemsTotal* que vimos anteriormente.

Esta misma técnica es la que he recomendado en el capítulo sobre Oracle, en relación con las tablas anidadas. Es recomendable guardar los cambios en la tabla maestra antes de añadir registros a la tabla anidada. También es bueno realizar un *Post* sobre la tabla maestra una vez que se ha añadido el registro en la tabla anidada:

```
void __fastcall TmodDatos::tbAnidadaAfterInsert(TDataSet *DataSet)
{
    Insertando = True;
}

void __fastcall TmodDatos::tbAnidadaAfterEdit(TDataSet *DataSet)
{
    Insertando = False;
}

void __fastcall TmodDatos::tbAnidadaAfterPost(TDataSet *DataSet)
{
    if (Insertando)
        tbMaestra->CheckBrowseMode();
}
```

En este ejemplo, la variable lógica *Insertando* debe haber sido definida en la sección **private** del módulo de datos.

Antes y después de la apertura de una tabla

Puede ser útil especificar acciones asociadas a la apertura y cierre de tablas. Si nuestra aplicación trabaja con muchas tablas, es conveniente que éstas se abran y cierren por demanda; si estas tablas representan objetos complejos, es posible expresar las dependencias entre tablas en los eventos *Before* y *AfterOpen*, y *Before* y *AfterClose*. En el ejemplo de la entrada de pedidos, la tabla de pedidos, *tbPedidos*, funciona en coordinación con la tabla de líneas de detalles, *tbLineas*. Supongamos, por un momento, que necesitamos también una tabla para resolver las referencias a clientes, *tbRefClientes*, y otra para las referencias a artículos, *tbRefArticulos*. Podemos entonces programar los siguientes métodos como respuesta a los eventos *BeforeOpen* y *AfterClose* de la tabla de pedidos:

```
void __fastcall TmodDatos::tbPedidosBeforeOpen(TDataSet *DataSet)
{
    tbRefClientes->Open();
    tbRefArticulos->Open();
    tbLineas->Open();
}
```

```

void __fastcall TmodDatos::tbPedidosAfterClose(TDataSet *DataSet)
{
    tbLineas->Close();
    tbRefArticulos->Close();
    tbRefClientes->Close();
}

```

De esta manera, las tres tablas dependientes pueden estar cerradas durante el diseño y carga de la aplicación, y ser activadas por demanda, en el momento en que se abra la tabla de pedidos.

El ejemplo que he utilizado quizás no sea el mejor. C++ Builder abre automáticamente las tablas de referencia cuando abre una tabla que tiene campos de este tipo. Sin embargo, al cerrar la tabla maestra, no cierra automáticamente las tablas de referencia.

Tirando de la cadena

Paradox, dBase y Access tienen un grave problema: si se cae el sistema después de una actualización, pueden estropearse las tablas irreversiblemente. Las modificaciones realizadas por una aplicación se guardan en *buffers* internos del BDE, desde donde son transferidas al disco durante los intervalos en que la aplicación está ociosa. Un corte de tensión, por ejemplo, puede dejar una modificación confirmada en un fichero índice, pero no en el fichero de datos principal, lo cual puede ser mortal en ocasiones.

La mejor cura consiste en transferir los *buffers* modificados en cuanto podamos, aún al coste de ralentizar la aplicación. C++ Builder ofrece el siguiente método, aplicable a los conjuntos de datos del BDE:

```

void __fastcall TBDEDataSet::FlushBuffers();

```

El mayor grado de seguridad se obtiene cuando llamamos a *FlushBuffers* inmediatamente después de cada modificación. Traducido al cristiano: en los eventos *AfterPost* y *AfterDelete* de las tablas y consultas actualizables:

```

void __fastcall TmodDatos::Table1AfterPost(TObject *Sender)
{
    static_cast<TBDEDataSet*>(Sender)->FlushBuffers();
}

```

Por supuesto, existen soluciones intermedias, como vaciar los *buffers* solamente después de terminar transacciones largas.

Los eventos de detección de errores

La política de control de errores de la VCL versión 1 estaba basada completamente en las excepciones, aplicando a rajatabla la Tercera Regla de Marteens:

*“Escriba la menor cantidad posible de instrucciones **try...catch**”*

Es decir, si se producía un error durante la grabación de un registro, se producía una excepción que iba abortando las funciones pendientes en la pila de ejecución del programa, y que terminaba su andadura en la instrucción de captura del ciclo de mensajes. Un cuadro de mensajes mostraba al usuario lo errado de sus planteamientos, y ¡vuelta a intentar la operación!

Bajo esta estrategia, ¿cómo mostrar mensajes más específicos? No nos interesa que el usuario de nuestro programa vea mensajes como: “Violación de unicidad”¹⁸. Nos interesa que el mensaje diga ahora “Código de cliente repetido”, y que en otra circunstancia diga “Código de artículo ya utilizado”. La única posibilidad que nos dejaba la VCL 1 era encerrar en instrucciones **try...catch** todas las llamadas que pudieran fallar, y esto no es siempre posible pues algunos métodos, como *Post*, son llamados implícitamente por otras operaciones.

A partir de la versión 2 de la VCL se introdujeron los llamados *eventos de detección de errores*, que se disparan cuando se produce un error en alguna operación de un conjunto de datos, pero antes de que se eleve la excepción asociada:

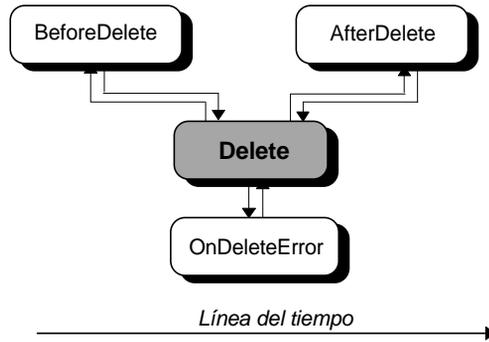
```
__property TDataSetErrorEvent OnEditError;
__property TDataSetErrorEvent OnPostError;
__property TDataSetErrorEvent OnDeleteError;
```

Por supuesto, estos eventos están disponibles en todas las versiones de C++ Builder. La declaración del tipo de estos eventos, en la enrevesada sintaxis de C++, es la siguiente:

```
typedef void __fastcall (__closure *TDataSetErrorEvent)
(TDataSet *DataSet, EDatabaseError *E, TDataAction &Action);
```

Estos eventos, cuando se disparan, lo hacen durante el “núcleo” de la operación asociada, después de que ha ocurrido el evento *Before*, pero antes de que ocurra el evento *After*. El siguiente diagrama representa el flujo de eventos durante la llamada al método *Delete*. Similar a este diagrama son los correspondientes a los métodos *Edit* y *Post*:

¹⁸ Suponiendo que los mensajes de error están traducidos, y no obtengamos: “*Key violation*”



Dentro del manejador del evento, podemos intentar corregir la situación que produjo el error, e indicar que la operación se reintente; para esto hay que asignar *daRetry* al parámetro *DataAction*. Podemos también dejar que la operación falle, y que C++ Builder muestre el mensaje de error; para esto se asigna a *DataAction* el valor *daFail*, que es el valor inicial de este parámetro. Por último, podemos elegir mostrar el mensaje dentro del manejador y que C++ Builder luego aborte la operación sin mostrar mensajes adicionales; esto se logra asignando *daAbort* al parámetro *DataAction*.

El ejemplo más sencillo de respuesta a un evento de detección de errores es la implementación de un ciclo de reintentos infinito, sin intervención del usuario, cuando se produce un error de bloqueo. Como estudiaremos en el capítulo sobre control de concurrencia, cuando se produce una situación tal, se dispara el evento *OnEditError*. Una posible respuesta es la siguiente:

```

void __fastcall TmodDatos::Table1EditError(TDataSet *DataSet,
    EDatabaseError *E, TDataAction &Action)
{
    // Esperar de 0.5 a 1 segundo
    Sleep(500 + random(500));
    // Reintentar
    Action = daRetry;
}
  
```

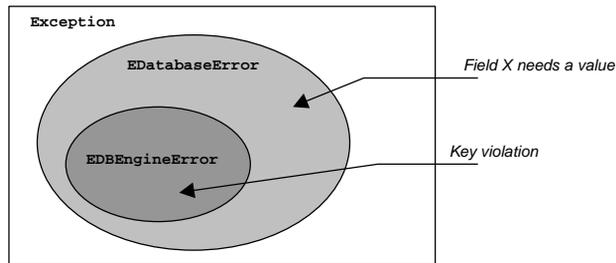
En este ejemplo, el reintento se produce después de una espera aleatoria que varía desde medio segundo hasta un segundo: de este modo se logra un tráfico menor en la red si se produce una colisión múltiple entre aplicaciones.

La estructura de la excepción *EDBEngineError*

En el caso del evento *OnEditError*, era relativamente fácil crear un manejador para el evento, pues los errores de edición tienen en su mayor parte una única causa: un bloqueo no concedido, en el caso de Paradox y dBase, y un registro eliminado por otro usuario, en cliente/servidor. Pero en el caso de los errores producidos por *Post* y *Delete*, las causas del error pueden ser variadas. En este caso, tenemos que aprovechar la

información que nos pasan en el parámetro *E*, que es la excepción que está a punto de generar el sistema. Y hay que aprender a interpretar el tipo *EDatabaseError*.

La clase de excepción *EDatabaseError* es la clase base de la jerarquía de excepciones de base de datos. Sin embargo, aunque existen algunos errores que se reportan directamente mediante esta excepción, la mayoría de los errores de bases de datos disparan una excepción de tipo *EDBEngineError*. La causa es sencilla: los errores del Motor de Datos pueden haber sido generados por el propio motor, pero también por un servidor SQL. Debemos conocer, entonces, el error original del servidor, pero también la interpretación del mismo que hace el BDE. La excepción generada por el BDE, *EDBEngineError*, tiene una estructura capaz de contener esta lista de errores. Por regla, si el error de base de datos es detectado por el BDE se lanza una excepción *EDBEngineError*; si el error es detectado por un componente de la VCL, se lanza un *EDatabaseError*.



Dentro del código fuente de la VCL, encontraremos a menudo una función *Check* definida en la unidad *DBTables*, cuyo objetivo es transformar un error del BDE en una excepción *EDBEngineError*. Por ejemplo:

```
// Código fuente en Delphi
procedure TDatabase.StartTransaction;
var
  TransHandle: HDBIXAct;
begin
  CheckActive;
  if not IsSQLBased and (TransIsolation <> tiDirtyRead) then
    DatabaseError(SLocalTransDirty, Self);
  Check(DbiBeginTran(FHandle, EXILType(FTransIsolation),
    TransHandle));
end;
```

Observe que la llamada a la función del API del BDE se realiza dentro de *Check*. De este modo, el resultado de *DbiBeginTran* es analizado por esta función: si es cero, no pasa nada, pero en caso contrario, se dispara la excepción. Observe también en el algoritmo anterior cómo se llama a una función *DatabaseError* cuando la VCL detecta por sí misma una condición errónea. Esta función es la encargada de disparar las

excepciones *EDatabaseError*. Existe también una versión *DatabaseErrorFmt*, en la que se permite pasar cadenas de formato, al estilo de la función *Format*.

Un objeto de la clase *EDBEngineError* tiene, además de las propiedades heredadas de *EDatabaseError*, las siguientes propiedades:

```
__property int ErrorCount;
__property TDBError *Errors[int Index];
```

El tipo *TDBError* corresponde a cada error individual. He aquí las propiedades disponibles para esta clase:

Propiedad	Significado
<i>ErrorCode</i>	El código del error, de acuerdo al BDE
<i>Category</i>	Categoría a la que pertenece el error
<i>SubCode</i>	Subcódigo del error
<i>NativeError</i>	Si el error es producido por el servidor, su código de error
<i>Message</i>	Cadena con el mensaje asociado al error.

NativeError corresponde al código *SqlCode* propio de cada servidor SQL. De todas estas propiedades, la más útil es *ErrorCode*, que es el código de error devuelto por el motor de datos. ¿Cómo trabajar con estos códigos? C++ Builder trae un ejemplo, *dberrors*, en el cual se explica cómo aprovechar los eventos de detección de errores. En el programa de demostración se utiliza un método bastante barroco para formar el código de error a partir de una dirección base y de unos valores numéricos bastante misteriosos. Curiosamente, todo este revuelo es innecesario, pues en la unidad *BDE* estos códigos de error ya vienen declarados como constantes que podemos utilizar directamente.

Para profundizar un poco más en el sistema de notificación de errores, cree una aplicación sencilla con una tabla, una rejilla y una barra de navegación. Luego asigne el siguiente manejador compartido por los tres eventos de errores, los correspondientes a *Post*, *Edit* y *Delete*:

```
void __fastcall TForm1::Table1PostError(TDataSet *TDataSet,
EDatabaseError *E, TDataAction &Action)
{
    AnsiString S;

    EDBEngineError *Err = dynamic_cast<EDBEngineError*>(E);
    if (Err)
    {
        for (int i = 0; i < Err->ErrorCount; i++)
        {
            if (i > 0) AppendStr(S, '\n');

```

```

TDBError *E = Err->Errors[i];
AppendStr(S, Format("%.4x (%d): %s",
    ARRAYOFCONST((E->ErrorCode, E->NativeError,
        E->Message))));
    }
DatabaseError(S, 0);
}
}

```

Por cada error que aparece en la lista, añadimos una línea al mensaje que estamos componiendo, mostrando el valor de *ErrorCode* en hexadecimal (pronto veremos por qué), el valor de *NativeError* y el mensaje específico de ese error. Cuando tenemos el mensaje, lanzamos una excepción de clase *EDatabaseError*. Dicho en otras palabras: no dejamos reaccionar al manejador del evento. Esta técnica es correcta, y se emplea con frecuencia cuando se quiere cambiar sencillamente el mensaje de error que se mostrará al usuario.

Ahora de lo que se trata es de hacer fallar a la aplicación por todas las causas que podamos, para ver cómo protesta C++ Builder. Por ejemplo, cuando se produce una violación de unicidad, éste es el resultado que he obtenido, haciendo funcionar a la aplicación contra una tabla de Paradox:

```
2601 (0): Key violation
```

El cero en *NativeError* indica que no es un error del servidor, pues no hay un servidor SQL en este caso. ¿Se ha dado cuenta de que he ignorado las dos propiedades *Category* y *SubCode* de los errores? Es que en este caso, la primera vale *0x26*, y la segunda *0x01*, expresados en notación hexadecimal. Con estos dos datos, usted puede bucear en la interfaz de la unidad *BDE*, para encontrar que corresponden a la constante *DBIERR_KEYVIOL*.

Sin embargo, si conecto la tabla a InterBase, éste es el error que obtengo:

```
2601 (0): Key violation
3303 (-803): Violation of PRIMARY or UNIQUE key constraint
"INTEG_56" on table "PARTS"
```

Primero el BDE ha detectado el error *-803* del servidor. ¿Recuerda que en el capítulo sobre *triggers* y procedimientos almacenados, al hablar de las excepciones mencionaba la variable *sqlcode*? Pues este código negativo corresponde al valor asumido por esta variable después del error. El problema es que no existe un estándar para los códigos de errores nativos en SQL, pero el SQL Link del BDE amablemente interpreta el error por nosotros, convirtiéndolo en nuestro conocido *0x2601*: "*Key violation*", que es lo que vería el usuario. Hay una regla que se puede inferir a partir de este ejemplo y otros similares: los códigos nativos del servidor vienen acompañando al *ErrorCode* *0x3303*, correspondiente a la constante simbólica *DBIERR_UNKNOWNSQL*.

Hagamos fallar una vez más al BDE. Esta vez he puesto una restricción en la propiedad *Constraints* de la tabla, y he intentado modificar un registro con valores incorrectos. Esto es lo que he obtenido:

```
2EC4 (0): Constraint failed. Expression:
2EAE (0): La cantidad de pedidos debe ser positiva.
```

Esta vez tenemos dos errores, y ninguno de ellos proviene del servidor, pues se trata de una restricción a verificar en el lado cliente. El primer error corresponde a la constante *DBIERR_USERCONSTREERR* (parece una maldición polaca). Y el segundo es *DBIERR_CONSTRAINTFAILED*. Lo curioso es que el algoritmo que muestra los mensajes de una excepción al usuario está preparado para ignorar el mensaje de la primera línea, y mostrar solamente el mensaje diseñado por el programador.

Aplicaciones de los eventos de errores

¿Qué podemos hacer en un evento de detección de errores? Hemos visto que en algunos casos se podía reintentar la operación, después de efectuar algunas correcciones, o de esperar un tiempo. Sin embargo, en la mayoría de los casos, no puede hacerse mucho. Tendremos que esperar al estudio de las actualizaciones en caché para disponer de herramientas más potentes, que nos permiten recuperarnos con efectividad de todo un rango de errores.

Sin embargo, la mayor aplicación de estos eventos es la *contextualización* de los mensajes de errores. Soy consciente de que acabo de escribir una palabra de 17 letras, ¿no debía haber dicho mejor *traducción*? Resulta que no. Un usuario intenta insertar un nuevo cliente, pero le asigna un nombre o un código ya existente. El BDE le responde enfadado: "*Key violation*". Pero usted servicialmente traduce el mensaje: "*Violación de clave*". Y a la mente del usuario, que no conoce a Codd y la historia de su perro, viene una serie desconcertante de asociaciones de ideas. Lo que teníamos que haber notificado era: "*Código o nombre de cliente ya existe*"¹⁹. Es decir, teníamos que adecuar el mensaje general a la situación particular en que se produjo.

Pero antes tendremos que simplificar un poco la forma de averiguar qué error se ha producido. La siguiente función nos va a ayudar en ese sentido, "aplanando" las excepciones de tipo *EDBEngineError* y produciendo un solo código de error que resume toda la información, o la parte más importante de la misma:

¹⁹ Y yo me pregunto: ¿por qué la mayoría de los mensajes en inglés y castellano prescinden de los artículos, las preposiciones y, en algunos casos, de los tiempos verbales? Yo Tarzán, tu Juana...

```

int GetBDEError(EDatabaseError *E)
{
    EDBEngineError *Err = dynamic_cast<EDBEngineError*>(E);
    if (Err)
        for (int I = 0; I < Err->ErrorCount; I++)
        {
            TDBError *dbe = Err->Errors[I];
            if (dbe->NativeError == 0)
                return dbe->ErrorCode;
        }
    return -1;
}

```

Si la excepción no es del BDE, la función devuelve -1, es decir, nada concreto. En caso contrario, husmeamos dentro de *Errors* buscando un error con el código nativo igual a cero: una interpretación del BDE. En tal caso, terminamos la función dejando como resultado ese primer error del BDE correspondiente a la parte cliente de la aplicación.

Ahora las aplicaciones de esta técnica. El siguiente método muestra cómo detectar los errores que se producen al borrar una fila de la cual dependen otras gracias a una restricción de integridad referencial. A diferencia de los ejemplos que hemos visto antes, este es un caso de detección de errores *a posteriori*:

```

void __fastcall TmodDatos::tbPedidosDeleteError(TDataSet *TDataSet,
    EDatabaseError *E, TDataAction &Action)
{
    if (GetBDEError(E) == DBIERR_DETAILRECORDSEXIST)
        if (MessageDlg("¿Eliminar también las líneas de detalles?",
            mtConfirmation, TMsgDlgButtons() << mbYes << mbNo, 0)
            == mrYes)
        {
            tbLineas->First();
            while (! tbLineas->Eof)
                tbLineas->Delete();
            // Reintentar el borrado en la tabla de pedidos
            Action = daRetry;
        }
    else
        // Fallar sin mostrar otro mensaje
        Action = daAbort;
}

```

Solamente hemos intentado solucionar el error cuando el código de error generado es *DBIERR_DETAILRECORDSEXIST*, que hemos encontrado en la unidad *BDE*.

Como último ejemplo, programaremos una respuesta que puede ser compartida por varias tablas en sus eventos *OnPostError*, y que se ocupa de la traducción genérica de los mensajes de excepción más comunes:

```

void __fastcall TmodDatos::ErrorDeGrabacion(TDataSet *DataSet,
    EDatabaseError *E, TDataAction &Action)
{
    TTable *T = static_cast<TTable*>(DataSet);

    switch (GetBDEError(E))
    {
        case DBIERR_KEYVIOL:
            DatabaseErrorFmt("Clave repetida en la tabla %s",
                ARRAYOFCONST((T->TableName)), 0);
            break;
        case DBIERR_FOREIGNKEYERR:
            DatabaseErrorFmt("Error en clave externa. Tabla: %s",
                ARRAYOFCONST((T->TableName)), 0);
            break;
    }
}

```

Observe que si *GetBDEError* no encuentra un código apropiado, este manejador de evento no hace nada, y el programa dispara la excepción original. Si el error ha sido provocado por el fallo de una restricción en el lado cliente, la función *ErrorDeGrabacion* dispara una nueva excepción con el mensaje personalizado introducido por el programador. Y si se trata de una violación de clave primaria, o de una integridad referencial, al menos se da un mensaje en castellano.

Una vez más, la orientación a objetos...

Los eventos de detección de errores nos muestran la forma correcta de entender la Programación Orientada a Objetos. Tomemos por caso la detección de los errores de violación de unicidad. Este es un error que se produce durante la ejecución del método *Post*. Por lo tanto, pudiéramos encerrar las llamadas al método *Post* dentro de instrucciones **try/catch**, y en cada caso tratar la excepción correspondiente. Este estilo de programación se orienta a la *operación*, más bien que al *objeto*, y nos fuerza a repetir el código de tratamiento de excepciones en cada caso en que la operación se emplea. Incluso, lo tendremos difícil si, como es el caso, la operación puede también producirse de forma implícita.

En cambio, utilizando los eventos de detección de errores, especificamos el comportamiento ante la situación de error una sola vez, asociando el código al objeto. De este modo, sea cual sea la forma en que se ejecute el método *Post*, las excepciones son tratadas como deseamos.

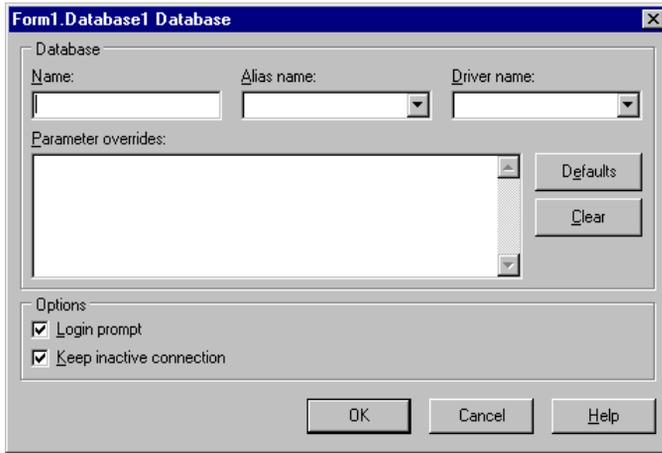
Bases de datos y transacciones

EN LA JERARQUÍA DE OBJETOS manejada por el BDE, las bases de datos y las sesiones ocupan los puestos más altos. En este capítulo estudiaremos la forma en que los componentes *TDatabase* controlan las conexiones a las bases de datos y la activación de transacciones. Dejaremos las posibilidades de los componentes de la clase *TSession* para el siguiente capítulo.

El componente *TDatabase*

Los componentes *TDatabase* de C++ Builder representan y administran las conexiones del BDE a sus bases de datos. Por ejemplo, este componente lleva la cuenta de las tablas y consultas activas en un instante determinado. Recíprocamente, las tablas y consultas están conectadas en tiempo de ejecución a un objeto *TDatabase*, que puede haber sido definido explícitamente por el programador, utilizando en tiempo de diseño el componente *TDatabase* de la paleta de componentes, o haber sido creado implícitamente por C++ Builder en tiempo de ejecución. Para saber si una base de datos determinada ha sido creada por el programador en tiempo de diseño o es una base de datos temporal creada por C++ Builder, tenemos la propiedad de tiempo de ejecución *Temporary*, en la clase *TDatabase*. Con las bases de datos se produce la misma situación que con los componentes de acceso a campos: que pueden definirse en tiempo de diseño o crearse en tiempo de ejecución con propiedades por omisión. Como veremos, las diferencias entre componentes *TDatabase* persistentes y dinámicos son mayores que las existentes entre ambos tipos de componentes de campos.

Las propiedades de un objeto *TDatabase* pueden editarse también mediante un cuadro de diálogo que aparece al realizar una doble pulsación sobre el componente:



Un objeto *TDatabase* creado explícitamente define siempre un alias local a la sesión a la cual pertenece. Básicamente, existen dos formas de configurar tal conexión:

- Crear un alias a partir de cero, siguiendo casi los mismos pasos que en la configuración del BDE, especificando el nombre del alias, el controlador y sus parámetros.
- Tomar como punto de partida un alias ya existente. En este caso, también se pueden alterar los parámetros de la conexión.

En cualquiera de los dos casos, la propiedad *IsSQLBased* nos dice si la base de datos está conectada a un servidor SQL o un controlador ODBC, o a una base de datos local.

Haya sido creado por la VCL en tiempo de ejecución, o por el programador en tiempo de diseño, un objeto de base de datos nos sirve para:

- Modificar los parámetros de conexión de la base de datos: contraseñas, conexiones establecidas, conjuntos de datos activos, etc.
- Controlar transacciones y actualizaciones en caché.

El control de transacciones se trata en el presente capítulo, y las actualizaciones en caché se dejan para más adelante.

Objetos de bases de datos persistentes

Comenzaremos con los objetos de bases de datos que el programador incluye en tiempo de diseño. La propiedad fundamental de estos objetos es *DatabaseName*, que corresponde al cuadro de edición *Name* del editor de propiedades. El valor alma-

cenado en *DatabaseName* se utiliza para definir un alias local a la aplicación. La forma en que se define este alias local depende de cuál de las dos propiedades, *AliasName* ó *DriverName*, sea utilizada por el programador. *AliasName* y *DriverName* son propiedades de uso mutuamente excluyente: si se le asigna algo a una, desaparece el valor almacenado en la otra. En este sentido se parecen al par *IndexName* e *IndexFieldNames* de las tablas. O al Yang y el Yin de los taoístas.

Si utilizamos *AliasName* estaremos definiendo un alias basado en otro alias existente. El objeto de base de datos puede utilizarse entonces para controlar las tablas pertenecientes al alias original. ¿Qué sentido tiene esto? La respuesta es que es posible modificar los parámetros de conexión del alias original. Esto quiere decir que podemos añadir parámetros nuevos en la propiedad *Params*. Esta propiedad, declarada de tipo *TStrings*, está inicialmente vacía para los objetos *TDatabase*. Se puede, por ejemplo, modificar un parámetro de conexión existente:

```
SQLPASSTHRUMODE=NOT SHARED
ENABLE SCHEMA CACHE=TRUE
```

El último parámetro permite acelerar las operaciones de apertura de tablas, y puede activarse cuando la aplicación no modifica dinámicamente el esquema relacional de la base de datos creando, destruyendo o modificando la estructura de las tablas. El significado del parámetro *SQLPASSTHRU MODE* ya ha sido estudiado en el capítulo sobre transacciones y control de concurrencia. Otro motivo para utilizar un alias local que se superponga sobre un alias persistente es la intercepción del evento de conexión a la base de datos (*login*). Pero muchas veces los programadores utilizan el componente *TDatabase* sólo para declarar una variable de este tipo que controle a las tablas pertinentes. Si está utilizando esta técnica con este único propósito, existen mejores opciones, como veremos dentro de poco.

La otra posibilidad es utilizar *DriverName*. ¿Recuerda cómo se define un alias con la configuración del BDE? Es el mismo proceso: *DatabaseName* indica el nombre del nuevo alias, mientras que *DriverName* especifica qué controlador, de los disponibles, queremos utilizar. Para configurar correctamente el alias, hay que introducir los parámetros requeridos por el controlador, y para esto utilizamos también la propiedad *Params*. De este modo, no necesitamos configurar alias persistentes para acceder a una base de datos desde un programa escrito en C++ Builder.

Cambiando un alias dinámicamente

Un buen ejemplo de aplicación que necesita utilizar alias locales, o de sesión, es aquella que, trabajando con tablas locales en formato Paradox o dBase, necesite cambiar periódicamente de directorio de trabajo. Por ejemplo, ciertas aplicaciones de contabilidad y gestión están diseñadas para trabajar con diferentes ejercicios o em-

presas, cuyas tablas se almacenan en distintos directorios del ordenador. Si se utilizan alias persistentes, es engorroso hacer uso de la utilidad de configuración del BDE, o de los métodos del componente *TSession* para definir o redefinir un alias persistente cada vez que tenemos que cambiar el conjunto de tablas con las cuales se trabaja.

Sin embargo, es relativamente fácil lograr este resultado si las tablas de la aplicación se conectan a un objeto *TDatabase* definido en tiempo de diseño, y este objeto define un alias local a la aplicación. Supongamos que la aplicación contiene, posiblemente en el módulo de datos, un objeto de tipo *TDatabase* con las siguientes propiedades:

Propiedad	Valor
<i>Name</i>	<i>Database1</i>
<i>DatabaseName</i>	<i>MiBD</i>
<i>DriverName</i>	<i>STANDARD</i>

Las tablas del módulo de datos, por supuesto, se conectarán por medio del alias *MiBD*, definido por este objeto. Para simplificar, supondré que dentro del módulo de datos solamente se ha colocado una tabla, *Table1*. Se puede asignar algún directorio inicial en la propiedad *Params* del componente *TDatabase*, incluyendo una línea con el siguiente formato:

```
PATH=C:\Archivos de programa\Contabilidad
```

El cambio de directorio debe producirse a petición del usuario de la aplicación; tras cada cambio, debe quedar grabado el camino al nuevo directorio dentro de un fichero de extensión *ini*. El siguiente método se encarga de cerrar la base de datos, cambiar el valor del parámetro *PATH* y reabrir la tabla, conectando de este modo la base de datos. Si todo va bien, se graba el nuevo directorio en un fichero de configuración de extensión *ini*:

```
void __fastcall TForm1::NuevoDirectorio(AnsiString ADir)
{
    DataModule1->Database1->Close();
    DataModule1->Database1->Params->Values["PATH"] = ADir;
    DataModule1->Table1->Open(); // Conecta también la BD
    std::auto_ptr<TIniFile> iniFile(
        new TIniFile(ChangeFileExt(Application->ExeName, ".INI")));
    iniFile->WriteString("Database", "Path", ADir);
}
```

Posiblemente, el método anterior se utilice después de que el usuario elija el directorio de trabajo mediante un cuadro de diálogo apropiado. Durante la carga del formulario se llama a *NuevoDirectorio* para utilizar el último directorio asignado, que debe encontrarse en el fichero de inicialización:

```

void __fastcall TForm1::FormCreate(TObject *Sender)
{
    AnsiString S;
    std::auto_ptr<TIniFile> iniFile(
        new TIniFile(ChangeFileExt(Application->ExeName, ".INI")));
    S = iniFile->ReadString("Database", "Path", "");
    if (! S.IsEmpty()) NuevoDirectorio(S);
}

```

Esta misma técnica puede aplicarse a otros tipos de controladores de bases de datos.

Bases de datos y conjuntos de datos

Es posible, en tiempo de ejecución, conocer a qué base de datos está conectado cierto conjunto de datos, y qué conjuntos de datos están activos y conectados a cierta base de datos. La clase *TDatabase* define las siguientes propiedades para este propósito:

```

__property int DataSetCount;
__property TDBDataSet* DataSets[int I];

```

Por ejemplo, se puede saber si alguno de los conjuntos de datos conectados a una base de datos contiene modificaciones en sus campos, sin que se le haya aplicado la operación *Post*:

```

bool HayModificaciones(TDatabase *ADatabase)
{
    int i = ADatabase.DataSetCount - 1;
    while (i >= 0 && ! ADatabase->DataSets[i]->Modified) i--;
    return (i >= 0);
}

```

En la sección anterior definimos un método *NuevoDirectorio* que cerraba una base de datos estándar, cambiaba su directorio asociado y volvía a abrirla, abriendo una tabla conectada a la misma. Ahora estamos en condiciones de generalizar este algoritmo, recordando qué conjuntos de datos estaban abiertos antes de cerrar la conexión para restaurarlos más adelante:

```

void __fastcall TForm1::NuevoDirectorio(TDatabase* ADB,
    const AnsiString ADir)
{
    std::auto_ptr<TList> Lista(new TList);
    // Recordar qué conjuntos de datos estaban abiertos
    for (int i = 0; i < ADB->DataSetCount; i++)
        Lista->Add(ADB->DataSets[i]);
    // Cambiar el directorio
    ADB->Close();
    ADB->Params->Values["PATH"] = ADir;
    ADB->Open();
}

```

```

// Reabrir los conjuntos de datos
for (int i = 0; i < Lista->Count; i++)
    static_cast<TDataSet*>(Lista->Items[i])->Open();
std::auto_ptr<TIniFile> iniFile(
    new TIniFile (ChangeFileExt(Application->ExeName, ".INI")));
iniFile->WriteString("Database", "Path", ADir);
}

```

Por otra parte, todo conjunto de datos activo tiene una referencia a su base de datos por medio de la propiedad *Database*. Antes mencionaba el hecho de que muchos programadores utilizan un componente *TDatabase* que definen sobre un alias persistente con el único propósito de tener acceso al objeto de bases de datos al que se conectan las tablas. La alternativa, mucho más eficiente, es declarar una variable de tipo *TDatabase* en la sección pública de declaraciones del módulo de datos, o en algún otro sitio conveniente, e inicializarla durante la creación del módulo:

```

class TDataModule1 : public TDataModule
{
    // ...
public:
    TDatabase *Database;
};

// ...

void __fastcall TDataModule1::DataModule2Create(TObject *Sender)
{
    Database = Table1->Database;
    if (! Database->IsSQLBased)
        Database->TransIsolation = tiDirtyRead;
}

```

La asignación realizada sobre la propiedad *TransIsolation* es bastante frecuente cuando se trata con tablas Paradox y dBase, y se quieren utilizar transacciones locales. Como ya explicamos en el capítulo 12, las bases de datos de escritorio solamente admiten el nivel más bajo de aislamiento. Por supuesto, cuando tenemos un componente *TDatabase* persistente, es preferible asignar el nivel de aislamiento directamente en el Inspector de Objetos.

Parámetros de conexión

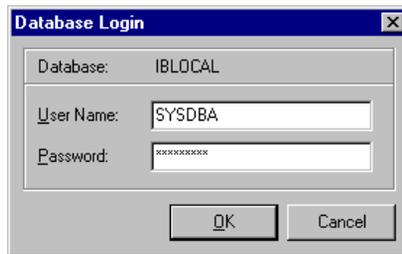
Hay aplicaciones que comienzan con todas sus tablas abiertas, y solamente las cierran al finalizar. Pero hay otras que van abriendo y cerrando tablas según sea necesario. ¿Qué debe suceder cuando todas las tablas han sido cerradas? Todo depende de la propiedad *KeepConnection*, del componente *TDatabase* al cual se asocian las tablas. Esta propiedad vale *True* por omisión, lo cual quiere decir que una vez establecida la conexión, ésta se mantiene aunque se cierren posteriormente todas las tablas. Si por el contrario, *KeepConnection* vale *False*, al cerrarse el último conjunto de datos activo de la base de datos, se desconecta la base de datos.

El problema es que una conexión a una base de datos consume recursos, especialmente si la base de datos se encuentra en un servidor. Típicamente, por cada usuario conectado, el software servidor debe asignar un proceso o hilo (*thread*), junto con memoria local para ese proceso. Así que en ciertos casos es conveniente que, al cerrarse la última tabla, se desconecte también la base de datos. Pero también sucede que el restablecimiento de la conexión es costoso, y si tenemos una base de datos protegida por contraseñas, el proceso de reapertura no es transparente para el usuario (a no ser que tomemos medidas). Por lo tanto, depende de usted lograr un buen balance entre estas dos alternativas.

La petición de contraseñas

Cuando se intenta establecer la conexión de un objeto *TDatabase* a su correspondiente base de datos, si la propiedad *LoginPrompt* del objeto es *True* y se trata de una base de datos SQL, el Motor de Datos debe pedir, de un modo u otro, la contraseña del usuario. Cómo se hace, depende de si hay algún método asociado con el evento *OnLogin* del objeto o no. Si el evento no está asignado, C++ Builder muestra un cuadro de diálogo predefinido, a través del cual se puede indicar el nombre del usuario y su contraseña. El nombre del usuario se inicializa con el valor extraído del parámetro *USER NAME* del alias:

```
Database1->Params->Values["USER NAME"]
```



Si, por el contrario, interceptamos el evento, es responsabilidad del método receptor asignar un valor al parámetro *PASSWORD* de la base de datos. Es bastante común interceptar este evento para mostrar un diálogo de conexión personalizado. Otro motivo para interceptar este evento puede ser la necesidad de quedarnos con el nombre del usuario, quizás para validar más adelante ciertas operaciones, o para llevar un registro de conexiones, si el sistema no lo hace automáticamente. Incluso, si queremos hacer trampas, es posible programar una especie de “caballo de Troya” para espiar las contraseñas, ya que también estarán a nuestra disposición; todo depende de la ética del programador.

La declaración del evento *OnLogin* es la siguiente:

```
typedef void __fastcall (__closure *TLoginEvent)
    (TDatabase *Database, TStrings *LoginParams);
```

Una posible respuesta a este evento puede ser:

```
void __fastcall TmodDatos::DatabaseLogin(TDatabase *Database,
    TStrings *LoginParams)
{
    if (FNombreUsuario.IsEmpty())
        if (dlgPassword->ShowModal() == mrOk)
        {
            FNombreUsuario = dlgPassword->Edit1->Text;
            FPassword = dlgPassword->Edit2->Text;
        }
    LoginParams->Values["USER NAME"] = FNombreUsuario;
    LoginParams->Values["PASSWORD"] = FPassword;
}
```

Este método asume que hay una sola base de datos en la aplicación. Estamos almacenando el nombre de usuario y su contraseña en los campos *FNombreUsuario* y *FPassword*, declarados como privados en la definición del módulo. Si es la primera vez que nos conectamos a la base de datos, aparece el diálogo mediante el cual pedimos los datos de conexión; aquí hemos nombrado *dlgPassword* a este formulario. A partir de la primera conexión, los datos del usuario no vuelven a necesitarse. Esto puede ser útil cuando intentamos mantener cerradas las conexiones inactivas (*KeepConnection* igual a *False*), pues es bastante engorroso que el usuario tenga que estar tecleando una y otra vez la contraseña cada vez que se reabre la base de datos.

Hay que tener cuidado, sin embargo, con la técnica anterior, pues no ofrece la posibilidad de verificar si los datos suministrados son correctos. De no serlo, cualquier intento posterior de conexión falla, pues ya están almacenados en memoria un nombre de usuario y contraseña no válidos.

En versiones anteriores de C++ Builder, el cuadro de diálogo de petición de contraseñas de la VCL era un “verdadero” cuadro de diálogo, cuya definición visual se almacenaba en un recurso asociado a la VCL y se creaba mediante una llamada a la función *CreateDialog* del API de Windows. En la versión 4 ya se ha convertido en un formulario con ejecución modal, por lo que si desea traducirlo al castellano solamente debe buscar el fichero *dfm* asociado.

Más motivos para desear cambiar la petición de contraseñas: Supongamos que estamos conectándonos a InterBase 5 o posterior. Recuerde que a partir de esta versión InterBase ofrece soporte para *roles*, y que estos roles se asumen durante la conexión a la base de datos. El diálogo de conexión estándar de la VCL no contempla la posibilidad de especificar un rol, así que en caso de querer aprovechar los roles, debemos crear nuestro propio formulario, con editores para el nombre de usuario, el rol y la

contraseña, y definir un manejador para el evento *OnLogin* que asigne los valores tecleados a los correspondientes parámetros del BDE:

```
void __fastcall TmodDatos::Database1Login(TDatabase *Database,
    TStrings *LoginParams)
{
    if (dlgPassword->ShowModal() == mrOk)
    {
        FNombreUsuario = dlgPassword->Edit1->Text;
        FRol = dlgPassword->Edit2->Text;
        FPassword = dlgPassword->Edit3->Text;
    }
    LoginParams->Values["USER NAME"] = FNombreUsuario;
    LoginParams->Values["ROLE NAME"] = FRol;
    LoginParams->Values["PASSWORD"] = FPassword;
}
```

El directorio temporal de Windows

El lector sabe que el parámetro *ENABLE SCHEMA CACHE* de los controladores SQL permite acelerar la conexión de una aplicación a un servidor, porque indica al BDE que almacene la información de esquema de las tablas en el cliente. Sabe también que el parámetro *SCHEMA CACHE DIR* sirve para indicar en qué directorio situar esta información de esquema. Sin embargo, es muy poco probable que alguien suministre un directorio estático para este parámetro, pues el directorio debe existir en el momento en que se realiza la conexión a la base de datos. Por ejemplo, ¿qué pasaría si quisiéramos que esta información se almacenara siempre en el directorio temporal de Windows? Pues que tendríamos problemas si la aplicación puede ejecutarse indistintamente en Windows NT o en Windows 95, ya que ambos sistemas operativos definen diferentes ubicaciones para sus directorios temporales.

La solución consiste en utilizar también el evento *OnLogin* para cambiar el directorio de la caché de esquemas *antes* de que se abra la base de datos:

```
void __fastcall TmodDatos::Database1Login(TDatabase *Database,
    TStrings *LoginParams)
{
    AnsiString S;
    S.SetLength(255);
    int L = GetTempPath(255, S.c_str());
    if (S.IsPathDelimiter(L)) L--;
    S.SetLength(L);
    Database1->Params->Values["ENABLE SCHEMA CACHE"] = "TRUE";
    Database1->Params->Values["SCHEMA CACHE DIR"] = S;
    // ...
}
```

El truco ha consistido en modificar el valor del parámetro directamente en la base de datos, no en el objeto *LoginParams* del evento.

También vale redefinir el método *Loaded* del módulo de datos, de forma similar a como hicimos para preparar las consultas explícitamente en el capítulo 24.

Compartiendo la conexión

Si un módulo de datos de C++ Builder 1 contenía un componente *TDatabase*, no era posible derivar módulos del mismo por medio de la herencia. La explicación es sencilla: un objeto *TDatabase* define, con su sola presencia, un alias local para la aplicación en la que se encuentra. Por lo tanto, si hubieran dos objetos de esta clase con iguales propiedades en dos módulos diferentes de la misma aplicación, y esto es lo que sucede cuando se utiliza la herencia visual, se intentaría definir el mismo alias local dos veces.

Existían dos soluciones para este problema. La primera era no utilizar objetos *TDatabase* persistentes en el módulo; si lo único que necesitábamos era acceso fácil al objeto *TDatabase* asociado a las tablas, para controlar transacciones, por ejemplo, se podía utilizar con la misma facilidad la propiedad *Database* de los conjuntos de datos. La otra solución consistía en situar el objeto de bases de datos en un módulo aparte de las tablas. En este caso, el módulo que contiene las tablas podía servir como clase base para la herencia, mientras que el módulo de la base de datos debía utilizarse directamente o ser copiado en nuestro proyecto.

Aislar la base de datos en un módulo separado puede seguir siendo conveniente por otros motivos. En el epígrafe anterior vimos una forma de evitar que el usuario telee innecesariamente sus datos cada vez que se inicia una conexión. Habíamos mencionado también el problema de validar la conexión, para reintentarla en caso de fallo. Si tenemos la base de datos aislada en un módulo, que se debe crear antes de los módulos que contienen las tablas, podemos controlar la conexión del siguiente modo, durante la creación del módulo:

```
void __fastcall TmodDatosDB::modDatosDBCreate(TObject *Sender)
{
    // Tres reintentos como máximo
    for (int Intentos = 0; Intentos < 3; Intentos++)
        try
        {
            Database1->Open();
            // Si todo va bien, nos vamos
            return;
        }
        catch(Exception&)
        {
            // Reiniciar los datos de usuario, y volver a probar
            FNombreUsuario = "";
            FPassword = "";
        }
}
```

```

// Si no se puede, terminar la aplicación
Application->Terminate();
}

```

La condición necesaria para que el código anterior funcione es que la base de datos esté cerrada en tiempo de diseño.

El problema expuesto anteriormente se resuelve a partir de C++ Builder 3 con una nueva propiedad: *HandleShared*. Si esta propiedad es *True*, dos bases de datos pueden tener el mismo nombre y compartir el mismo *handle* del BDE. Si colocamos un *TDatabase* en un módulo, basta con activar esta propiedad para poder derivar módulos por herencia sin ningún tipo de problemas.

Control explícito de transacciones

Como ya el lector se habrá dado cuenta, el modo de trabajo habitual de C++ Builder considera que cada actualización realizada sobre una tabla está aislada lógicamente de las demás posibles actualizaciones. Para base de datos locales, esto quiere decir sencillamente que no hay transacciones involucradas en el asunto. Para un sistema SQL, las cosas son distintas.

El objeto encargado de activar las transacciones explícitas es el componente *TDatabase*. Los tres métodos que ofrece *TDatabase* para el control explícito de transacciones son:

```

void __fastcall TDatabase::StartTransaction();
void __fastcall TDatabase::Commit();
void __fastcall TDatabase::Rollback();

```

Después de una llamada a *Rollback* es aconsejable realizar una operación *Refresh* sobre todas las tablas abiertas de la base de datos, para releer los datos y actualizar la pantalla. Considerando que la base de datos lleva cuenta de los conjuntos de datos activos, se puede automatizar esta operación:

```

void CancelarCambios(TDatabase* ADatabase)
{
    ADatabase->Rollback();
    for (int i = ADatabase->DataSetCount - 1; i >= 0; i--)
        ADatabase->DataSets[i]->Refresh();
}

```

La propiedad *InTransaction*, disponible en tiempo de ejecución, nos avisa si hemos iniciado alguna transacción sobre la base de datos activa:

```

void __fastcall TForm1::TransaccionClick(TObject *Sender)
{
    IniciarTransaccion1->Enabled = ! Database1->InTransaction;
}

```

```

    ConfirmarTransaccion1->Enabled = Databasel->InTransaction;
    CancelarTransaccion1->Enabled = Databasel->InTransaction;
}

```

Una transferencia bancaria que se realice sobre bases de datos de escritorio, por ejemplo, podría programarse del siguiente modo:

```

// Iniciar la transacción
Databasel->StartTransaction();
try
{
    TLocateOptions Opt;
    if (! Table1->Locate("Apellidos", "Einstein", Opt))
        DatabaseError("La velocidad de la luz no es un límite", 0);
    Table1->Edit();
    Table1->FieldValues["Saldo"] =
        Table1->FieldValues["Saldo"] - 10000;
    Table1->Post();
    if (! Table1->Locate("Apellidos", "Newton", Opt))
        DatabaseError("No todas las manzanas caen al suelo", 0);
    Table1->Edit();
    Table1->FieldValues["Saldo"] =
        Table1->FieldValues["Saldo"] + 10000;
    Table1->Post();
    // Confirmar la transacción
    Databasel->Commit();
}
catch(Exception&)
{
    // Cancelar la transacción
    Databasel->Rollback();
    Table1->Refresh();
    throw;
}

```

Observe que la presencia de la instrucción **throw** al final de la cláusula **catch** garantiza la propagación de una excepción no resuelta, de modo que no quede enmascarada; de esta forma no se viola la Tercera Regla de Marteens.

Entrada de datos y transacciones

Se puede aprovechar el carácter atómico de las transacciones para automatizar el funcionamiento de los diálogos de entrada de datos que afectan simultáneamente a varias tablas de una misma base de datos, del mismo modo que lo hemos logrado con las actualizaciones sobre una sola tabla. Si el lector recuerda, para este tipo de actualizaciones utilizábamos la siguiente función, que llamábamos desde la respuesta al evento *OnCloseQuery* del cuadro de diálogo:

```

bool PuedoCerrar(TForm *AForm, TDataSet *DS)
{
    if (AForm->ModalResult == mrOk)
        DS->Post();
}

```

```

else if (! DS->Modified ||
    Application->MessageBox("¿Desea abandonar los cambios?",
        "Atención", MB_ICONQUESTION | MB_YESNO) == IDYES)
    DS->Cancel();
else
    return False;
return True;
}

```

La generalización del algoritmo de cierre del cuadro de diálogo es inmediata. Observe el uso que se hace de *CheckBrowseMode*, para garantizar que se graben los cambios pendientes.

```

bool PuedoCerrarTrans(TForm *AForm,
    TDBDataSet* const* DataSets, int DataSets_size)
{
    if (AForm->ModalResult == mrOk)
    {
        for (int i = 0; i <= DataSets_size; i++)
            DataSets[i]->CheckBrowseMode();
        DataSets[0]->Database->Commit();
    }
    else if (Application->MessageBox("¿Desea abandonar los cambios?",
        "Atención", MB_ICONQUESTION | MB_YESNO) == IDYES)
    {
        for (int i = 0; i <= DataSets_size; i++)
            DataSets[i]->Cancel();
        DataSets[0]->Database->Rollback();
    }
    else
        return False;
    return True;
}

```

Esta función se debe llamar desde el evento *OnCloseQuery* del diálogo de la siguiente manera:

```

void __fastcall TdlgPedidos::FormCloseQuery(TObject *Sender,
    var CanClose: Boolean);
{
    CanClose = PuedoCerrarTrans(this, OPENARRAY(TDBDataSet*,
        (modDatos->tbPedidos, modDatos->tbDetalles)));
}

```

Ahora hay que llamar explícitamente a *StartTransaction* antes de comenzar la edición o inserción:

```

void __fastcall TPrincipal::AltaPedidos(TObject *Sender)
{
    modDatos->tbPedidos->Database->StartTransaction();
    modDatos->tbPedidos->Append();
    dlgPedidos->ShowModal();
}

```

Es un poco incómodo tener que iniciar explícitamente la transacción cada vez que se active el cuadro de diálogo. También hemos perdido la posibilidad de detectar fácilmente si se han realizado modificaciones en algunas de la tablas involucradas; es posible que algunos de los cambios realizados hayan sido enviados con el método *Post* a la base de datos, como sucede frecuentemente en la edición *master/detail*. De esta forma, si activamos por error el formulario de entrada de datos y pulsamos la tecla ESC inmediatamente, obtenemos una inmerecida advertencia sobre las consecuencias de abandonar nuestros datos a su suerte.

Más adelante estudiaremos las *actualizaciones en caché*, un mecanismo que nos ayudará, entre otras cosas, a resolver éstos y otros inconvenientes menores.

Sesiones

SI SEGUIMOS ASCENDIENDO EN LA JERARQUÍA de organización de objetos del BDE, pasaremos de las bases de datos a las sesiones. El estudio de estos componentes es de especial importancia cuando necesitamos utilizar varios hilos en una misma aplicación, pero también cuando deseamos extraer información del BDE y del esquema de una base de datos, como demostraremos en la aplicación que desarrollaremos al final del presente capítulo.

¿Para qué sirven las sesiones?

El uso de sesiones en C++ Builder nos permite lograr los siguientes objetivos:

- Cada sesión define un usuario diferente que accede al BDE. Si dentro de una aplicación queremos sincronizar acciones entre procesos, sean realmente concurrentes o no, necesitamos sesiones.
- Las sesiones nos permiten administrar desde un mismo sitio las conexiones a bases de datos de la aplicación. Esto incluye la posibilidad de asignar valores por omisión a las propiedades de las bases de datos.
- Las sesiones nos dan acceso a la configuración del BDE. De este modo podemos administrar los alias del BDE y extraer información de esquemas de las bases de datos.
- Mediante las sesiones, podemos controlar el proceso de conexión a tablas Paradox protegidas por contraseñas.

En las versiones de 16 bits del BDE, anteriores a C++ Builder, sólo era necesaria una sesión por aplicación, porque podía ejecutarse un solo hilo por aplicación. En la primera versión de la VCL, por ejemplo, la clase *TSession* no estaba disponible como componente en la Paleta, y para tener acceso a la única sesión del programa teníamos la variable global *Session*. En estos momentos, además de poder crear sesiones adicionales en tiempo de diseño, seguimos teniendo la variable *Session*, que esta vez se refiere a la sesión por omisión. También se ha añadido la variable global *Sessions*, de la

clase *TSessionList*, que permite el acceso a todas las sesiones existentes en la aplicación.

Tanto *Sessions*, como *Session*, están declaradas en la unidad *DBTables*, y son creadas automáticamente por el código de inicialización de esta unidad en el caso de que sea mencionada por alguna de las unidades de su proyecto.

Especificando la sesión

Los componentes *TSession* tienen una propiedad llamada *SessionName*, de tipo *Ansi-String*, que le sirve al BDE para identificar las sesiones activas. No pueden existir simultáneamente dos sesiones que compartan el mismo nombre dentro de un mismo proceso. La sesión por omisión, a la que se refiere la variable global *Session*, se define con el nombre de *Default*.

Cada componente *TDatabase* posee también una propiedad *SessionName* que debe coincidir con el nombre de alguna de las sesiones del proyecto. Por supuesto, el valor de esta propiedad para un *TDatabase* recién creado es *Default*, con lo que se indica que la conexión se realiza mediante la sesión por omisión. Como explicamos en el capítulo anterior, todo *TDatabase* crea un alias “de sesión” dentro de la aplicación a la cual pertenece. Ahora podemos ser más precisos: este alias temporal realmente pertenece a la sesión a la cual se asocia el componente. Esto quiere decir que podemos tener dos componentes *TDatabase* dentro de la misma aplicación con el mismo valor en sus propiedades *DatabaseName*, aún cuando *HandleShared* sea *False* para ambos.

En consecuencia, todos los conjuntos de datos derivados de *TDBDataSet* también contienen una propiedad *SessionName*. Cuando desplegamos en el Inspector de Objetos la lista de valores asociada a la propiedad *DatabaseName* de una tabla o consulta, solamente veremos los alias persistentes *más* los alias de sesión definidos para la sesión a la cual está asociado el conjunto de datos.

Cada sesión es un usuario

Hemos explicado que cada sesión define un acceso diferente al BDE, como si fuera un usuario distinto. La consecuencia más importante de esto es que el BDE levanta barreras de contención entre estos diferentes usuarios. Por ejemplo, si en una sesión se abre una tabla en modo exclusivo, dentro de la misma sesión se puede volver a abrir la tabla en este modo, pues la sesión no se bloquea a sí misma. Lo mismo ocurre con cualquier posible bloqueo a nivel de registro. Es fácil realizar una demostración práctica de esta peculiaridad. Sobre un formulario vacío coloque dos objetos de tipo *TSession* y configúrelos del siguiente modo:

Propiedad	Primera sesión	Segunda sesión
<i>Name</i>	<i>Session1</i>	<i>Session2</i>
<i>SessionName</i>	<i>S1</i>	<i>S2</i>

Traiga también un par de tablas, con las siguientes propiedades:

Propiedad	Primera tabla	Segunda tabla
<i>Name</i>	<i>Table1</i>	<i>Table2</i>
<i>DatabaseName</i>	<i>dbdemos</i>	<i>dbdemos</i>
<i>TableName</i>	<i>biolife.db</i>	<i>biolife.db</i>
<i>Exclusive</i>	<i>true</i>	<i>true</i>

Por último, sitúe dos botones sobre el formulario, con las etiquetas “*Una sesión*” y “*Dos sesiones*”. El primer botón intentará abrir las dos tablas en exclusiva durante la misma sesión; el segundo hará lo mismo, asignando primero diferentes sesiones. La respuesta a estos botones será compartida:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    Table1->Close();
    Table2->Close();
    Table1->SessionName = "S1";
    if (Sender == Button1)
        Table2->SessionName = "S1";
    else
        Table2->SessionName = "S2";
    Table1->Open();
    Table2->Open();
}
```

El resultado, por supuesto, será que el primer botón podrá ejecutar su código sin problemas, mientras que el segundo botón fallará en su intento.

Ha sido fundamental que la tabla del ejemplo perteneciera a una base de datos de escritorio. La mayoría de los sistemas SQL ignoran la propiedad *Exclusive* de las tablas.

El inicio de sesión y la inicialización del BDE

La inicialización del Motor de Datos de Borland por la VCL corre a cargo del componente *TSession*. Cada vez que se va a ejecutar un método de esta clase, la implementación verifica en primer lugar que la propiedad *Active* sea *True*, es decir, que la sesión esté iniciada. Si no lo está, se inicializa el BDE, lo cual quiere decir que se cargan las estructuras de las DLLs del Motor de Datos en memoria. Si es la primera vez que se inicializa una sesión en esa máquina en particular, las DLLs necesarias se

cargan en memoria por primera vez. En caso contrario, se inicializa un nuevo cliente o instancia de las DLLs.

Comprender cómo funciona la inicialización del BDE es importante, pues durante la misma la VCL especifica en qué idioma deben estar los mensajes del Motor de Datos. El siguiente método ha sido extraído del código fuente de la VCL (en Delphi) y muestra cómo la sesión inicializa ritualmente el BDE:

```

procedure TSession.InitializeBDE;
var
    Status: DBIResult;
    Env: DbiEnv;
    ClientHandle: hDBIObj;
    SetCursor: Boolean;
begin
    SetCursor := (GetCurrentThreadID = MainThreadID)
        and (Screen.Cursor = crDefault);
    if SetCursor then
        Screen.Cursor := crHourGlass;
    try
        FillChar(Env, SizeOf(Env), 0);
        StrPLCopy(Env.szLang, SIDAPILangID, SizeOf(Env.szLang) - 1);
        Status := DbiInit(@Env);
        if (Status <> DBIERR_NONE)
            and (Status <> DBIERR_MULTIPLEINIT) then
            Check(Status);
        Check(DbiGetCurrSession(FHandle));
        if DbiGetObjFromName(objCLIENT, nil, ClientHandle) = 0 then
            DbiSetProp(ClientHandle, Integer(c1SQLRESTRICT), GDAL);
        if IsLibrary then
            DbiRegisterCallback(nil, cbDETACHNOTIFY, 0, 0, nil,
                DLLDetachCallback);
    finally
        if SetCursor and (Screen.Cursor = crHourGlass) then
            Screen.Cursor := crDefault;
    end;
end;

```

Como podemos ver, el método gira alrededor de la llamada a la función *DbiInit*, para la cual preparamos una variable *Env*, de tipo *DbiEnv*. Esta función puede detectar si no es la primera vez que se inicializa el BDE (código de retorno *DBIERR_MULTIPLEINIT*); si tal cosa sucede, la VCL lo pasa por alto. La parte que nos interesa en este momento es la inicialización de la variable de entorno, *Env*. El único campo que nos tomamos la molestia de modificar es *szLang*, en el cual se copia el valor de la constante de cadenas de recursos *SIDAPILangID*. En una versión de C++ Builder para inglés, esta cadena contiene el valor "0009", que como podemos verificar en el fichero *winnt.h*, corresponde al inglés.

Esta inicialización instruye al BDE para que busque sus mensajes dentro de los recursos almacenados en la biblioteca dinámica *idr20009.dll*. Si queremos nuestros mensajes en castellano, por ejemplo, tenemos que crear un fichero *idr2000a.dll*, y traducir la cadena de recurso *SIDAPILangID* de la VCL al valor "000A". Pero si desea-

mos ahorrarnos el último paso, podemos también inicializar el BDE en nuestra aplicación antes de que el código de sesiones tenga su oportunidad. En cualquier momento antes de la carga de los formularios y módulos de datos de la aplicación podemos insertar las siguientes instrucciones:

```
DBIEnv Env;
memset(Env, 0, sizeof(Env));
strncpy(Env.szLang, "000A", sizeof(Env.szLang) - 1);
Check(DbiInit(&Env));
```

Es posible pasar el puntero nulo como parámetro de la función *DbiInit*. En tal caso, el lenguaje con el que se inicializa el BDE corresponde al identificador almacenado en la clave de registro siguiente:

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Borland\Database Engine\RESOURCE]
```

Sesiones e hilos paralelos

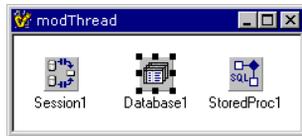
La principal aplicación de estas propiedades de la sesión es poder realizar operaciones de bases de datos en distintos hilos de la misma aplicación; cada hilo enchufa sus componentes de bases de datos por medio de una sesión diferente. Los servidores de automatización y las extensiones ISAPI/NSAPI para servidores de Internet, que estudiaremos en capítulos posteriores, permiten que varios clientes se conecten a la misma instancia de la aplicación. A cada cliente se le asigna un hilo diferente, por lo que es esencial utilizar sesiones para evitar conflictos entre las peticiones y modificaciones de datos. Todo esto lo veremos en su momento.

Ahora bien, ¿es apropiado utilizar hilos en aplicaciones clientes “normales”? El ejemplo más socorrido en los libros de C++ Builder es el de una aplicación MDI que abre una ventana hija basada en el resultado de la ejecución de una consulta. Como la ejecución de la consulta por el servidor (o por el intérprete local) puede tardar, para que el usuario no pierda el control de la interfaz de la aplicación, la apertura de la consulta se efectúa en un hilo separado. La técnica es correcta, y los motivos impecables. Pero yo nunca haría tal disparate en una aplicación real, pues cada ventana lanzada de esta manera consumiría una conexión a la base de datos, que es un recurso generalmente limitado y costoso. Aún después de haber terminado la ejecución del hilo que abre la consulta, el objeto *TQuery* sigue conectado a una sesión separada, como si hubiera un Dr. Jekyll y un Mr. Hide dentro de mi ordenador personal.

Se me ocurre, sin embargo, un ejemplo ligeramente diferente en el que sí es recomendable utilizar un hilo en paralelo. Sustituya en el párrafo anterior la palabra “consulta” por “procedimiento almacenado”. Supongamos que nuestra base de datos cliente/servidor tiene definido un procedimiento que realiza una operación de man-

tenimiento larga y costosa, y que ese procedimiento debemos lanzarlo desde la aplicación. En principio, el procedimiento no devuelve nada importante. Si utilizamos la técnica de lanzamiento convencional, tenemos que esperar a que el servidor termine para poder continuar con la aplicación. Y esta espera es la que podemos evitar utilizando hilos y sesiones. ¿Qué diferencia hay con respecto al ejemplo de la consulta? Una fundamental: que cuando termina la ejecución del procedimiento podemos desconectar la sesión adicional, mientras que no podemos hacer lo mismo con la consulta hasta que no cerremos la ventana asociada.

Iniciamos una aplicación. Ponga en la ventana principal una rejilla conectada a una tabla perteneciente a una base de datos cliente/servidor. Ahora cree un módulo de datos, llámelo *modThread* y coloque en él los siguientes tres objetos:



En el objeto *Session1* solamente es necesario asignar un nombre en *SessionName*, digamos que sea *S1*. Este mismo nombre debe copiarse en la propiedad homónima del *TDatabase*. Configure, además, este componente para que podamos conectarnos a la misma base de datos que estamos explorando en la ventana principal. Finalmente, cambie también *SessionName* en el procedimiento almacenado, y engánchelo a algún procedimiento almacenado de la base de datos cuya ejecución requiera bastante tiempo. Y muy importante: ¡deje inactivos a todos los objetos! No queremos que esta sesión esté abierta desde que arranque la aplicación.

Vamos ahora a programar el hilo que se encargará de ejecutar el procedimiento. Ejecute *File | New* para obtener el diálogo del Depósito de Objetos, y realice una doble pulsación en el icono *Thread object*:



Este experto crea, en una unidad aparte, un esqueleto de clase que debemos modificar del siguiente modo:

```
class TSPThread : public TThread
{
protected:
    void __fastcall Execute();
};
```

```
public:
    __fastcall TSPThread();
};
```

Hemos añadido un constructor al objeto. He aquí el cuerpo de los métodos:

```
__fastcall TSPThread::TSPThread() :
    TThread(True) // Crear un hilo "suspendido"
{
    FreeOnTerminate = True;
    Resume(); // Continuar la ejecución
}

void __fastcall TSPThread::Execute()
{
    modThread->Database1->Open();
    try
    {
        modThread->StoredProc1->ExecProc();
    }
    __finally
    {
        modThread->Session1->Close();
    }
}
```

El constructor crea inicialmente el hilo en estado “suspendido”, es decir, no comienza inmediatamente su ejecución. Antes de lanzarlo, asigna *True* a la propiedad *FreeOnTerminate*, para que la memoria del objeto *TSPThread* sea liberada al finalizar la ejecución del hilo. Lo que haga el hilo estará determinado por el contenido del método *Execute*. En éste se accede a los objetos necesarios (sesión, base de datos, procedimiento almacenado) mediante la variable global *modThread*; recuerde que los hilos comparten el mismo espacio de memoria dentro de una aplicación. He abierto explícitamente la base de datos antes de ejecutar el procedimiento, y después me he asegurado de que se cierra la sesión (y con ella la base de datos). Quizás usted tenga que retocar un poco el código para que no se vuelva a pedir el nombre de usuario y su contraseña al abrirse la base de datos.

Con este objeto a nuestra disposición, lo único que tiene que hacer la ventana principal para ejecutar el procedimiento almacenado en segundo plano es lo siguiente:

```
void __fastcall TForm1::Consolidacion1Click(TObject *Sender)
{
    new TSPThread;
}
```

Al crearse el objeto del hilo, automáticamente se inicia su ejecución. Recuerde que la última instrucción del constructor es *Resume*. La destrucción del objeto creado es automática, y ocurre cuando el procedimiento almacenado ha finalizado su acción, y se ha roto la segunda conexión a la base de datos.

Información sobre esquemas

La clase *TSession* tiene métodos para extraer la información sobre esquemas de las bases de datos registradas por el BDE. Para comenzar por la cima de la jerarquía, tenemos *GetDriverNames*, para recuperar los nombres de controladores instalados:

```
void __fastcall TSession::GetDriverNames(TStrings *Lista);
```

Este método, y la mayoría de los que siguen a continuación que devuelven una lista de nombres, vacían primero la lista de cadenas antes de asignar valores. En este caso, en *Lista* queda la lista de controladores registrados en el BDE; el controlador *STANDARD* se refiere a Paradox y dBase. Una vez que tenemos un nombre de controlador podemos averiguar sus parámetros:

```
void __fastcall TSession::GetDriverParams(
    const AnsiString Controlador, TStrings *Lista);
```

Para obtener la lista de bases de datos y alias, se utilizan *GetDatabaseNames* y *GetAliasName*. La diferencia entre ambos métodos es que el primero devuelve, además de los alias persistentes, los alias locales declarados mediante objetos *TDatabase*; el segundo se limita a los alias persistentes. Tenemos además las funciones *GetAliasDriverName* y *GetAliasParams* para extraer la información asociada a un alias determinado:

```
void __fastcall TSession::GetDatabaseNames(TStrings *Lista);
void __fastcall TSession::GetAliasNames(TStrings *Lista);
AnsiString __fastcall TSession::GetAliasDriverName(
    const AnsiString Alias);
void __fastcall TSession::GetAliasParams(const AnsiString Alias,
    TStrings *Lista);
```

Una vez que tenemos un alias en la mano, podemos averiguar qué tablas existen en la base de datos asociada. Para esto utilizamos el método *GetTableNames*:

```
void __fastcall TSession::GetTableNames(const AnsiString Alias,
    const AnsiString Patron, bool Extensiones, bool TablasDeSistema,
    TStrings *Lista);
```

El parámetro *Patron* permite filtrar las bases de datos; la cadena vacía se utiliza para seleccionar todas las tablas. El parámetro *Extensiones* sirve para incluir o eliminar las extensiones de ficheros en los sistemas de bases de datos locales. Por último, *TablasDelSistema* se utiliza en las bases de datos SQL para incluir o descartar las tablas que el sistema de bases de datos crea automáticamente.

Del mismo modo, para una base de datos SQL se puede utilizar el siguiente método que devuelve los procedimientos almacenados definidos:

```
void __fastcall TSession::GetStoredProcNames(const AnsiString Alias,
    TStringList *Lista);
```

El MiniExplorador de Bases de Datos

El ejemplo siguiente muestra una forma sencilla de llenar un árbol con la información de alias y tablas existentes. Necesitamos un formulario con un componente *TTreeView*, de la página *Win32* de la paleta, alineado a la izquierda, un cuadro de lista, *TListBox*, que ocupe el resto del formulario, y un objeto *TImageList*, también de la página *Win32*, para que contenga los iconos que vamos a mostrar al lado de los alias y las tablas. Para inicializar este último objeto, pulsamos dos veces sobre el mismo. En el editor que aparece, utilizamos el botón *Add* para cargar un par de imágenes en el control. Utilizaremos la primera para representar los alias, dejando la segunda para las tablas.

Ahora debemos interceptar el evento *OnCreate* del formulario, para inicializar la lista de alias presentes en el ordenador:

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    std::auto_ptr<TStringList> List(new TStringList);
    Session->GetDatabaseNames(List.get());
    for (int i = 0; i < List->Count; i++)
    {
        TTreeNode* AliasPtr =
            TreeView1->Items->Add(NULL, List->Strings[i]);
        AliasPtr->ImageIndex = 0;
        TreeView1->Items->AddChild(AliasPtr, "");
    }
}
```

He utilizado el truco de añadir un hijo ficticio al nodo alias, para que aparezca el botón de expansión. Solamente cuando se pulse por primera vez este botón, se buscarán las tablas pertenecientes al alias, con el objetivo de minimizar el tiempo de carga de la aplicación. La expansión del nodo se realiza en respuesta al evento *OnExpanding* del visualizador de árboles:

```
void __fastcall TForm1::TreeView1Expanding(TObject *Sender,
    TTreeNode *Node, bool &AllowExpansion)
{
    if (Node->Data != NULL) return;
    Node->DeleteChildren();
    std::auto_ptr<TStringList> List(new TStringList);
    Session->GetTableNames(Node->Text, "", False, False, List.get());
    for (int i = 0; i < List->Count; i++)
    {
        TTreeNode* t =
            TreeView1->Items->AddChild(Node, List->Strings[i]);
        t->ImageIndex = 1;
    }
}
```

```

        t->SelectedIndex = 1;
    }
    Node->Data = Node;
}

```

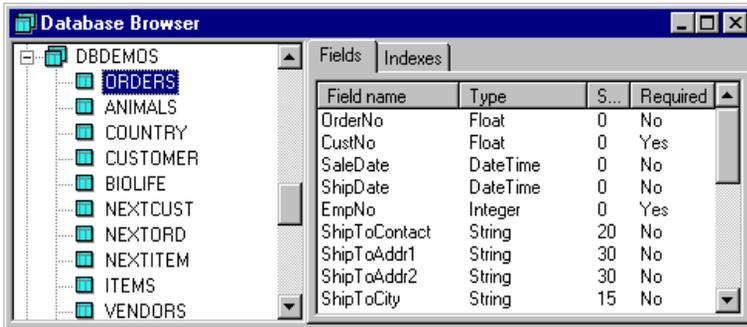
Estoy utilizando otro truco “sucio” para saber si un nodo ha sido expandido o no. Los nodos de árboles tienen una propiedad *Data*, de tipo **void***, en la cual podemos guardar la información que se nos antoje. Para este ejemplo, si *Data* contiene el puntero vacío estamos indicando que el nodo aún no ha sido expandido; cuando el nodo se expanda, *Data* pasará a apuntar al propio nodo.

Por último, cuando seleccionemos un nodo alias, extraeremos la información del mismo y la visualizaremos en el cuadro de listas de la derecha:

```

void __fastcall TForm1::TreeView1Change(TObject *Sender,
    TTreeNode *Node)
{
    if (! Node->Parent)
    {
        ListBox1->Items->BeginUpdate();
        try
        {
            Session->GetAliasParams(Node->Text, ListBox1->Items);
            ListBox1->Items->Insert(0,
                "DRIVER=" + Session->GetAliasDriverName(Node->Text));
        }
        finally
        {
            ListBox1->Items->EndUpdate();
        }
    }
}

```



Observe el uso del método *BeginUpdate* para evitar el parpadeo provocado por la actualización de la pantalla mientras insertamos cadenas en el control.

Gestión de alias a través de *TSession*

También se pueden crear alias y modificar parámetros de alias existentes mediante los objetos de sesión. Para añadir nuevos alias, utilizamos los siguientes métodos:

```
void __fastcall TSession::AddAlias(const AnsiString Nombre,
    const AnsiString CtrlIdor, TStrings *Lista);
void __fastcall TSession::AddStandardAlias(const AnsiString Nombre,
    const AnsiString Dir, const AnsiString CtrlIdor);
```

AddAlias es el método más general para añadir un nuevo alias, mientras que *AddStandardAlias* simplifica las cosas cuando queremos crear un alias para bases de datos locales. El tipo de alias que se cree, persistente o local, depende de la propiedad *ConfigMode* de la sesión; esta variable puede asumir los valores *cmPersistent*, *cmSession* y *cmAll*. El último valor es el valor por omisión. Si queremos que el alias creado sea local, debemos utilizar el valor *cmSession*.

Para modificar o eliminar un alias existente, se pueden utilizar los siguientes métodos:

```
void __fastcall TSession::ModifyAlias(const AnsiString Alias,
    TStrings *Parametros);
void __fastcall TSession::DeleteAlias(const AnsiString Alias);
```

Por último, para guardar la configuración y que los cambios sean permanentes hay que utilizar el siguiente método:

```
void __fastcall TSession::SaveConfigFile();
```

En nuestro mini-explorador de bases de datos añadimos un par de comandos de menú: *Crear alias* y *Eliminar alias*. Para eliminar un alias utilizamos el siguiente procedimiento:

```
void __fastcall TForm1::miDeleteAliasClick(Sender: TObject);
{
    TTreeNode* t = TreeView1->Selected;
    if (t->Parent == NULL &&
        MessageDlg("Eliminando el alias " + t->Text +
            ".\n¿Está seguro?", mtConfirmation,
            TMsgDlgButtons() << mbYes << mbNo, 0) == mrYes)
    {
        Session->DeleteAlias(t->Text);
        delete t;
    }
}
```

Para crear un nuevo alias, necesitaremos una segunda ventana en la aplicación, a la cual nombraremos *dlgNewAlias*. Esta ventana, configurada como diálogo, contiene los siguientes objetos:

Componente	Función
<i>Edit1</i>	Contiene el nombre del nuevo alias.
<i>ComboBox1</i>	De estilo <i>csDropDownList</i> , contiene las cadenas <i>PARADOX</i> y <i>DBASE</i> . Es el nombre del controlador por omisión.
<i>DirectoryListBox1</i>	Para seleccionar un directorio.

Necesitamos también botones para aceptar y cancelar la ejecución del diálogo. La respuesta al comando de creación de alias es la siguiente:

```
void __fastcall TForm1::miNewAliasClick(TObject *Sender)
{
    if (dlgNewAlias->ShowModal() == mrOk)
    {
        Session->AddStandardAlias(dlgNewAlias->Edit1->Text,
            dlgNewAlias->DirectoryListBox1->Directory,
            dlgNewAlias->ComboBox1->Text);
        TTreeNode *AliasPtr = TreeView1->Items->Add(NULL,
            dlgNewAlias->Edit1->Text);
        AliasPtr->ImageIndex = 0;
        TreeView1->Items->AddChild(AliasPtr, "");
    }
}
```

Para simplificar la explicación, hemos creado un alias para el controlador estándar. La creación de alias para otros controladores no plantea mayores dificultades.

Directorios privados, de red y contraseñas

Como si las sesiones no hicieran bastante ya, también se ocupan de la configuración de los directorios *NetDir* y *PrivateDir* del Borland Database Engine, y de la gestión de contraseñas de usuarios de Paradox. Ya hemos visto, al configurar el BDE, la función de estos directorios. Sepa ahora que puede cambiarlos desde un programa Delphi utilizando objetos *TSession*, si no se han efectuado aún conexiones a bases de datos. Para cambiar estas propiedades es útil el evento *OnStartup*, que se dispara justamente antes de iniciar la sesión.

En cuanto a las contraseñas de las tablas Paradox, el mecanismo de gestión de las mismas es diferente al de las bases de datos SQL típicas. Aquí, las contraseñas se definen a nivel de tabla, no de la base de datos. Por lo tanto, la contraseña se pide al intentar abrir una tabla protegida mediante este recurso. Este es el comportamiento de C++ Builder por omisión, y no hay que programar nada especial para trabajar con este tipo de tablas. Sin embargo, por causa de que las contraseñas son locales a tablas, si tenemos un par de tablas protegidas por la misma contraseña, tendremos que teclearla dos veces para abrir las dos tablas. Esto puede ser bastante engorroso, por lo cual el BDE permite almacenar a nivel de sesión un conjunto de contraseñas permitidas. El cuadro de apertura para tablas con contraseña permite almacenar en la se-

sión actual la contraseña suministrada. De este modo, si se guarda la contraseña durante la apertura de la primera tabla, ésta no será solicitada al abrir la siguiente tabla.

Los siguientes métodos de las sesiones trabajan sobre la lista de contraseñas disponibles en la sesión:

```
void __fastcall TSession::AddPassword(const AnsiString Password);
void __fastcall TSession::RemovePassword(const AnsiString Password);
void __fastcall TSession::RemoveAllPasswords();
```

Si queremos saltarnos el diálogo de petición de contraseñas de la VCL, podemos realizar una llamada a *AddPassword* en cualquier momento previo a la apertura de la tabla protegida. Por lo general, el mejor momento para esto es durante el evento *OnPassword* del componente *TSession*. Este evento se dispara cuando se produce un error al abrir una tabla por no disponer de los suficientes derechos. *OnPassword* pertenece al tipo de eventos que controlan un bucle de reintentos; para esto cuenta con un parámetro lógico *Continue*, con el cual podemos controlar el fin del bucle:

```
void __fastcall TForm1::Session1Password(TObject *Sender,
    bool &Continue)
{
    AnsiString S;
    if (InputQuery("Dis-moi, miroir magique...",
        "¿Cuál es el mejor lenguaje de la Galaxia?", S) &&
        S.Pos("Visual") == 0)
    {
        Session1->AddPassword("BuenChico");
        Continue = True;
    }
}
```


Actualizaciones en caché

LAS ACTUALIZACIONES EN CACHÉ SON UN RECURSO de las versiones de 32 bits del BDE para aumentar el rendimiento de las transacciones en entornos cliente/servidor. Los conjuntos de datos de C++ Builder vienen equipados con una propiedad, *CachedUpdates*, que decide si los cambios efectuados en el conjunto de datos son grabados inmediatamente en la base de datos o si se almacenan en memoria del ordenador cliente y se envían en bloque al servidor, a petición del programa cliente, en un momento dado.

En este capítulo estudiaremos las características básicas de las actualizaciones en caché, y cómo se pueden aplicar a la automatización de los procesos de entrada de datos. Al final, veremos cómo aprovechar esta técnica para mejorar el tratamiento de consultas en entornos cliente/servidor y para minimizar el impacto sobre el usuario de los bloqueos optimistas.

¿Caché para qué?

¿Qué nos aporta este intrincado mecanismo? En primer lugar, mediante este recurso una transacción que requiera interacción con el usuario puede hacerse más corta. Y, como hemos explicado antes, una transacción mientras más breve, mejor. Por otra parte, las actualizaciones en caché pueden disminuir drásticamente el número de paquetes enviados por la red. Cuando no están activas las actualizaciones en caché, cada registro grabado provoca el envío de un paquete de datos. Cada paquete va precedido de cierta información de control, que se repite para cada envío. Además, estos paquetes tienen un tamaño fijo, y lo más probable es que se desaproveche parte de su capacidad. También se benefician aquellos sistemas SQL que utilizan internamente técnicas pesimistas de bloqueos para garantizar las lecturas repetibles. En este caso, los bloqueos impuestos están activos mucho menos tiempo, durante la ejecución del método *ApplyUpdates*. De este modo, se puede lograr en cierto modo la simulación de un mecanismo optimista de control de concurrencia.

Las ventajas mencionadas se aplican fundamentalmente a los entornos cliente/servidor. Pero también es correcto el uso de actualizaciones en caché para bases de

datos locales. Esta vez la razón es de simple conveniencia para el programador, y tiene que ver nuevamente con la modificación e inserción de objetos complejos, representados en varias tablas. Como vimos con anterioridad, se pueden utilizar las transacciones para lograr la atomicidad de estas operaciones. Pero para programar una transacción necesitamos iniciarla y confirmarla o deshacerla explícitamente, mientras que, como veremos, una actualización en caché no necesita ser iniciada: en cada momento existe sencillamente un conjunto de actualizaciones realizadas desde la última operación de confirmación. Todo lo cual significa menos trabajo para el programador.

En particular, si intentamos utilizar transacciones sobre bases de datos locales, tenemos que enfrentarnos al límite de bloqueos por tabla (100 para dBase y 255 para Paradox), pues las transacciones no liberan los bloqueos hasta su finalización. Si en vez de utilizar directamente las transacciones locales utilizamos actualizaciones en caché, la implementación de las mismas por el BDE sobrepasa esta restricción escalando automáticamente el nivel de los bloqueos impuestos.

La historia, sin embargo, no acaba aquí. Al seguir estando disponibles los datos originales de un registro después de una modificación o borrado, tenemos a nuestro alcance nuevas posibilidades: la selección de registros de acuerdo a su estado de actualización y la cancelación de actualizaciones registro a registro. Incluso podremos implementar algo parecido a los borrados lógicos de dBase.

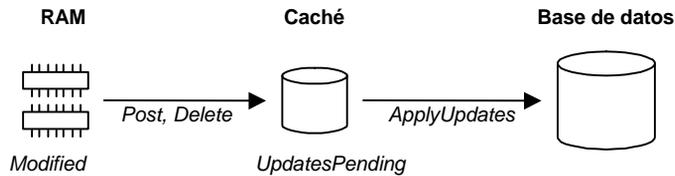
Me permitiré una breve digresión lingüística sobre nuestro neologismo “caché”. Como sucede con muchas de las palabras incorporadas al castellano que están relacionadas con la informática, esta incorporación ha sido incorrecta. El problema es que pronunciamos “caché”, acentuando la palabra en la última sílaba. En inglés, sin embargo, esta palabra se pronuncia igual que *cash*, y su significado es esconder; en particular, puede significar un sitio oculto donde se almacenan provisiones. Es curioso que, en francés, existe la expresión *cache-cache* que se pronuncia como en inglés y quiere decir “el juego del escondite” (*cache* es ocultar).

Activación de las actualizaciones en caché

La activación del mecanismo se logra asignando el valor *True* a la propiedad *Cached-Updates* de las tablas o consultas. El valor de esta propiedad puede cambiarse incluso estando la tabla activa. Como las actualizaciones en caché utilizan internamente transacciones para su confirmación, cuando se efectúan sobre tablas locales la propiedad *TransIsolation* de la correspondiente base de datos debe valer *tiDirtyRead*.

Se puede conocer si existen actualizaciones en la caché, pendientes de su confirmación definitiva, utilizando la propiedad de tipo lógico *UpdatesPending* para cada tabla o

consulta. Observe que la propiedad *UpdatesPending* solamente informa acerca de las actualizaciones realizadas con *Post* y *Delete*; si la tabla se encuentra en alguno de los modos de edición *dsEditModes* y se han realizado asignaciones a los campos, esto no se refleja en *UpdatesPending*, sino en la propiedad *Modified*, como siempre.



La activación de la caché de actualizaciones es válida únicamente para el conjunto de datos implicado. Si activamos *CachedUpdates* para un objeto *TTable*, y creamos otro objeto *TTable* que se refiera a la misma tabla física, los cambios realizados en la primera tabla no son visibles desde la segunda hasta que no se realice la confirmación de los mismos.

Una vez que las actualizaciones en caché han sido activadas, los registros del conjunto de datos se van cargando en el cliente en la medida en que el usuario va leyendo y realizando modificaciones. Es posible, sin embargo, leer el conjunto de datos completo desde un servidor utilizando el método *FetchAll*:

```
void __fastcall TDBDataSet::FetchAll();
```

De esta forma, se logra replicar el conjunto de datos en el cliente. No obstante, este método debe usarse con cuidado, debido al gran volumen de datos que puede duplicar.

Confirmación de las actualizaciones

Existen varios métodos para la confirmación definitiva de las actualizaciones en caché. El más sencillo es el método *ApplyUpdates*, que se aplica a objetos de tipo *TDatabase*. *ApplyUpdates* necesita, como parámetro, la lista de tablas en las cuales se graban, de forma simultánea, las actualizaciones acumuladas en la caché:

```
void __fastcall TDatabase::ApplyUpdates(
    TDBDataSet* const * DataSets,
    const int DataSets_size);
```

Un detalle interesante, que nos puede ahorrar código: si la tabla a la cual se aplica el método *ApplyUpdates* se encuentra en alguno de los estados de edición, se llama de forma automática al método *Post* sobre la misma. Esto implica también que *Apply-*

Updates graba, o intenta grabar, las modificaciones pendientes que todavía residen en el *buffer* de registro, antes de confirmar la operación de actualización.

A un nivel más bajo, los conjuntos de datos tienen implementados los métodos *ApplyUpdates* y *CommitUpdates*; la igualdad de nombres entre los métodos de los conjuntos de datos y de las bases de datos puede confundir al programador nuevo en la orientación a objetos. Estos son métodos sin parámetros:

```
void __fastcall TDataSet::ApplyUpdates();
void __fastcall TDataSet::CommitUpdates();
```

ApplyUpdates, cuando se aplica a una tabla o consulta, realiza la primera fase de un protocolo en dos etapas; este método es el encargado de grabar físicamente los cambios de la caché en la base de datos. La segunda fase es responsabilidad de *CommitUpdates*. Este método limpia las actualizaciones ya aplicadas que aún se encuentran en la caché. ¿Por qué necesitamos un protocolo de dos fases? El problema es que, si realizamos actualizaciones sobre varias tablas, y pretendemos grabarlas atómicamente, tenemos que enfrentarnos a la posibilidad de errores de grabación, ya sean provocados por el control de concurrencia o por restricciones de integridad. Por lo tanto, en el algoritmo de confirmación se han desplazado las operaciones falibles a la primera fase, la llamada a *ApplyUpdates*; por el contrario, *CommitUpdates* no debe fallar nunca, a pesar de Murphy.

La división en dos fases la aprovecha el método *ApplyUpdates* de la clase *TDatabase*. Para aplicar las actualizaciones pendientes de una lista de tablas, la base de datos inicia una transacción e intenta llamar a los métodos *ApplyUpdates* individuales de cada conjunto de datos. Si falla alguno de éstos, no pasa nada, pues la transacción se deshace y los cambios siguen residiendo en la memoria caché. Si la grabación es exitosa en conjunto, se confirma la transacción y se llama sucesivamente a *CommitUpdates* para cada conjunto de datos. El esquema de la implementación de *ApplyUpdates* es el siguiente:

```
StartTransaction();           // this = la base de datos
try
{
    for (int i = 0; i <= DataSets_size; i++)
        DataSets[i]->ApplyUpdates(); // Pueden fallar
    Commit();
}
catch(Exception&)
    Rollback();
    throw;                     // Propagar la excepción
}
for (int i = 0; i <= DataSets_size; i++)
    DataSets[i]->CommitUpdates(); // Nunca fallan
```

Es recomendable llamar siempre al método *ApplyUpdates* de la base de datos para confirmar las actualizaciones, en vez de utilizar los métodos de los conjuntos de

datos, aún en el caso de una sola tabla o consulta. No obstante, es posible aprovechar estos procedimientos de más bajo nivel en circunstancias especiales, como puede suceder cuando queremos coordinar actualizaciones en caché sobre dos bases de datos diferentes.

Por último, una advertencia: como se puede deducir de la implementación del método *ApplyUpdates* aplicable a las bases de datos, las actualizaciones pendientes se graban en el orden en que se pasan las tablas dentro de la lista de tablas. Por lo tanto, si estamos aplicando cambios para tablas en relación *master/detail*, hay que pasar primero la tabla maestra y después la de detalles. De este modo, las filas maestras se graban antes que las filas dependientes. Por ejemplo:

```
Database1->ApplyUpdates(
    OPENARRAY(TDBDataSet*, (tbPedidos, tbDetalles)));
```

Debemos tener en cuenta que, en bases de datos cliente/servidor, los *triggers* asociados a una tabla se disparan cuando se graba la caché, no en el momento en que se ejecuta el *Post* sobre la tabla correspondiente. Lo mismo sucede con las restricciones de unicidad y de integridad referencial. Las restricciones **check**, sin embargo, pueden duplicarse en el cliente mediante las propiedades *Constraints*, del conjunto de datos, y *CustomConstraint* ó *ImportedConstraint* a nivel de campos.

Marcha atrás

En contraste, no existe un método predefinido que descarte las actualizaciones pendientes en todas las tablas de una base de datos. Para descartar las actualizaciones pendientes en caché, se utiliza el método *CancelUpdates*, aplicable a objetos de tipo *TDBDataSet*. Del mismo modo que *ApplyUpdates* llama automáticamente a *Post*, si el conjunto de datos se encuentra en algún estado de edición, *CancelUpdates* llama implícitamente a *Cancel* antes de descartar los cambios no confirmados.

El siguiente procedimiento muestra una forma sencilla de descartar cambios en una lista de conjuntos de datos:

```
void DescartarCambios(TDBDataSet* const* DataSets,
    int DataSets_size)
{
    for (int i = 0; i <= DataSets_size; i++)
        DataSets[i]->CancelUpdates();
}
```

También se pueden cancelar las actualizaciones para registros individuales. Esto se consigue con el método *RevertRecord*, que devuelve el registro a su estado original.

El estado de actualización

La función *UpdateStatus* de un conjunto de datos indica el tipo de la última actualización realizada sobre el registro activo. *UpdateStatus* puede tomar los siguientes valores:

Valor	Significado
<i>usUnmodified</i>	El registro no ha sufrido actualizaciones
<i>usModified</i>	Se han realizado modificaciones sobre el registro
<i>usInserted</i>	Este es un registro nuevo
<i>usDeleted</i>	Este registro ha sido borrado (ver más adelante)

La forma más sencilla de comprobar el funcionamiento de esta propiedad es mostrar una tabla con actualizaciones en caché sobre una rejilla, e interceptar el evento *OnDrawColumnCell* de la rejilla para mostrar de forma diferente cada tipo de registro. Por ejemplo:

```
void __fastcall TForm1::DBGrid1DrawColumnCell(TObject *Sender,
const TRect& Rect, int DataCol, TColumn *Column,
TGridDrawState State)
{
    TDBGrid *grid = static_cast<TDBGrid*>(Sender);
    TFontStyles FS;
    switch (grid->DataSource->DataSet->UpdateStatus())
    {
        case usModified: FS << fsBold; break;
        case usInserted: FS << fsItalic; break;
        case usDeleted: FS << fsStrikeOut; break;
    }
    grid->Canvas->Font->Style = FS;
    grid->DefaultDrawColumnCell(Rect, DataCol, Column, State);
}
```

También puede aprovecharse la función para deshacer algún tipo de cambios en la tabla:

```
void __fastcall TForm1::DeshacerInsercionesClick(TObject *Sender)
{
    Table1->DisableControls();
    AnsiString BM = Table1->Bookmark;
    try
    {
        Table1->First();
        while (! Table1->Eof)
            if (Table1->UpdateStatus() == usInserted)
                Table1->RevertRecord();
            else
                Table1->Next();
    }
    finally
    {
        Table1->Bookmark = BM;
    }
}
```

```

        Table1->EnableControls();
    }
}

```

En el siguiente epígrafe veremos un método más sencillo de descartar actualizaciones selectivamente.

El filtro de tipos de registros

¿Qué sentido tiene el poder marcar una fila de una rejilla como borrada, si nunca podemos verla? Pues sí se puede ver. Para esto, hay que modificar la propiedad *UpdateRecordTypes* del conjunto de datos en cuestión. Esta propiedad es un conjunto que puede albergar las siguientes constantes:

Valor	Significado
<i>rtModified</i>	Mostrar los registros modificados
<i>rtInserted</i>	Mostrar los nuevos registros
<i>rtDeleted</i>	Mostrar los registros eliminados
<i>rtUnModified</i>	Mostrar los registros no modificados

Inicialmente, esta propiedad contiene el siguiente valor, que muestra todos los tipos de registros, con excepción de los borrados:

```
TUpdateRecordTypes() << rtModified << rtInserted << rtUnModified
```

Si por algún motivo asignamos el conjunto vacío a esta propiedad, el valor de la misma se restaura a este valor por omisión:

```

Table1->UpdateRecordTypes = TUpdateRecordTypes();
if (Table1->UpdateRecordTypes == TUpdateRecordTypes() <<
    rtModified << rtInserted << rtUnModified)
    ShowMessage("UpdateRecordTypes restaurado al valor por omisión");

```

Combinemos ahora el uso de los filtros de tipos de registros con este nuevo método, para recuperar de forma fácil todos los registros borrados cuya eliminación no ha sido aún confirmada:

```

void __fastcall TForm1::bnRecuperarClick(TObject *Sender)
{
    TUpdateRecordTypes URT = Table1->UpdateRecordTypes;
    Table1->UpdateRecordTypes = TUpdateRecordTypes() << rtDeleted;
    try {
        Table1->First();
        while (! Table1->Eof)
            Table1->RevertRecord();
    }
}

```

```

    __finally {
        Table1->UpdateRecordTypes = URT;
    }
}

```

No hace falta avanzar el cursor hasta el siguiente registro después de recuperar el actual, porque automáticamente el registro recuperado desaparece de la vista del cursor. Para completar el ejemplo, sería necesario restaurar la posición inicial de la tabla, y desactivar temporalmente la visualización de los datos de la misma; esto queda como ejercicio para el lector.

Un ejemplo integral

El siguiente ejemplo integra las distintas posibilidades de las actualizaciones en caché de modo tal que el lector puede verificar el funcionamiento de cada una de ellas. Necesitamos un formulario con una tabla, *Table1*, una fuente de datos *DataSource1*, una rejilla de datos y una barra de navegación. Da lo mismo la tabla y la base de datos que elijamos; lo único que necesitamos es asignar *True* a la propiedad *CachedUpdates* de la tabla.

Entonces necesitamos un menú. Estas son las opciones que incluiremos:

Caché	Ver
Grabar	Originales
Cancelar	Modificados
Cancelar actual	Nuevos
	Borrados

Para hacer más legible el código que viene a continuación, he renombrado coherentemente las opciones de menú; así, la opción *Caché*, cuyo nombre por omisión sería *Cach1*, se ha transformado en *miCache* (*mi* = *menu item*).

En primer lugar, daremos respuesta a los tres comandos del primer submenú:

```

void __fastcall TForm1::miAplicarClick(TObject *Sender)
{
    if (! Table->Database->IsSQLBased)
        Table1->Database->TransIsolation = tiDirtyRead;
    Table1->Database->ApplyUpdates((TDBDataSet**) &Table1, 0);
}

void __fastcall TForm1::miCancelarClick(TObject *Sender)
{
    Table1->CancelUpdates();
}

```

```

void __fastcall TForm1::miCancelarActualClick(TObject *Sender)
{
    Table1->RevertRecord();
}

```

La primera línea en el primer método sólo es necesaria si se ha elegido como tabla de prueba una tabla Paradox o dBase. Ahora necesitamos activar y desactivar las opciones de este submenú; esto lo hacemos en respuesta al evento *OnClick* de *miCache*:

```

void __fastcall TForm1::miCacheClick(TObject *Sender)
{
    miAplicar->Enabled = Table1->UpdatesPending;
    miCancelar->Enabled = Table1->UpdatesPending;
    miCancelarActual->Enabled =
        Table1->UpdateStatus() != usUnModified;
}

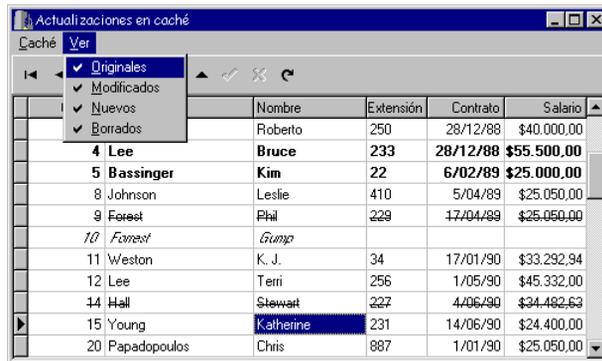
```

Luego creamos un manejador compartido para los cuatro comandos del menú *Ver*:

```

void __fastcall TForm1::ComandosVer(TObject *Sender)
{
    TMenuItem* Command = static_cast<TMenuItem*>(Sender);
    Command->Checked = ! Command->Checked;
    TUpdateRecordTypes URT;
    if (miOriginales->Checked) URT << rtUnmodified;
    if (miModificados->Checked) URT << rtModified;
    if (miNuevos->Checked) URT << rtInserted;
    if (miBorrados->Checked) URT << rtDeleted;
    Table1->UpdateRecordTypes = URT;
}

```



Por último, debemos actualizar las marcas de verificación de estos comandos al desplegar el submenú al que pertenecen:

```

void __fastcall TForm1::miVerClick(TObject *Sender)
{
    TUpdateRecordTypes URT = Table1->UpdateRecordTypes;
    miOriginales->Checked = URT.Contains(rtUnmodified);
    miModificados->Checked = URT.Contains(rtModified);
    miNuevos->Checked = URT.Contains(rtInserted);
}

```

```

        miBorrados->Checked = URT.Contains(rtDeleted);
    }

```

Si lo desea, puede incluir el código de personalización de la rejilla de datos que hemos analizado antes, para visualizar el estado de actualización de cada registro mostrado.

El Gran Final: edición y entrada de datos

Podemos automatizar los métodos de entrada de datos en las aplicaciones que tratan con objetos complejos, del mismo modo que ya lo hemos hecho con la edición de objetos representables en una sola tabla. En el capítulo sobre bases de datos y transacciones habíamos desarrollado la función *PuedoCerrarTrans*, que cuando se llamaba desde la respuesta al evento *OnCloseQuery* de la ventana de entrada de datos, se ocupaba de guardar o cancelar automáticamente las grabaciones efectuadas sobre un conjunto de tablas. Esta era la implementación lograda entonces:

```

bool PuedoCerrarTrans(TForm *AForm,
    TDBDataSet* const* DataSets, int DataSets_size)
{
    if (AForm->ModalResult == mrOk)
    {
        for (int i = 0; i <= DataSets_size; i++)
            DataSets[i]->CheckBrowseMode();
        DataSets[0]->Database->Commit();
    }
    else if (Application->MessageBox("¿Desea abandonar los cambios?",
        "Atención", MB_ICONQUESTION | MB_YESNO) == IDYES)
    {
        for (int i = 0; i <= DataSets_size; i++)
            DataSets[i]->Cancel();
        DataSets[0]->Database->Rollback();
    }
    else
        return False;
    return True;
}

```

El uso de esta rutina se complicaba, porque antes de ejecutar el cuadro de diálogo había que iniciar manualmente una transacción, además de poner la tabla principal en modo de inserción o edición. Teníamos el inconveniente adicional de no saber si se habían realizado modificaciones durante la edición, al menos sin programación adicional; esto causaba que la cancelación del diálogo mostrase cada vez un incómodo mensaje de confirmación.

En cambio, si utilizamos actualizaciones en caché obtenemos los siguientes beneficios:

- No hay que llamar explícitamente a *StartTransaction*.
- Combinando las propiedades *Modified* y *UpdatesPending* de los conjuntos de datos involucrados en la operación, podemos saber si se han realizado modificaciones, ya sea a nivel de campo o de caché.
- El uso de *StartTransaction* disminuye la posibilidad de acceso concurrente, pues por cada registro modificado durante la transacción, el sistema coloca un bloqueo que solamente es liberado cuando cerramos el cuadro de diálogo. Con las actualizaciones en caché, la transacción se inicia y culmina durante la respuesta a los botones de finalización del diálogo.

Esta es la función que sustituye a la anterior:

```
bool PuedoCerrar(TForm *AForm,
    TDBDataSet* const* DataSets, int DataSets_size)
{
    // Verificamos si hay cambios en caché
    bool Actualizar = False;
    for (int i = 0; i <= DataSets_size; i++)
        if (DataSets[i]->UpdatesPending || DataSets[i]->Modified)
            Actualizar = True;
    // Nos han pedido que grabemos los cambios
    if (AForm->ModalResult == mrOk)
        DataSets[0]->Database->ApplyUpdates(DataSets, DataSets_size);
    // Hay que deshacer los cambios
    else if (! Actualizar ||
        Application->MessageBox("¿Desea abandonar los cambios?",
            "Atención", MB_ICONQUESTION | MB_YESNO) == IDYES)
        for (int i = 0; i <= DataSets_size; i++)
            // CancelUpdates llama a Cancel si es necesario
            DataSets[i]->CancelUpdates();
    // El usuario se arrepiente de abandonar los cambios
    else
        return False;
    return True;
}
```

Se asume que todas las tablas están conectadas a la misma base de datos; la base de datos en cuestión se extrae de la propiedad *Database* de la primera tabla de la lista.

Si, por ejemplo, el formulario de entrada de datos *entPedidos* realiza modificaciones en las tablas *tbPedidos*, *tbDetalles*, *tbClientes* y *tbArticulos* (*orders*, *items*, *customer* y *parts*) pertenecientes al módulo de datos *modDatos* y conectadas a la misma base de datos, podemos asociar el siguiente método al evento *OnCloseQuery* de la ventana en cuestión:

```
void __fastcall entPedidos::entPedidosCloseQuery(TObject *Sender,
    bool &CanClose)
{
    CanClose = PuedoCerrar(this, OPENARRAY(TDBDataSet*,
        (modDatos->tbPedidos, modDatos->tbDetalles,
        modDatos->tbClientes, modDatos->tbArticulos)));
}
```

El orden en que se pasan las tablas a esta función es importante. Tenga en cuenta que para grabar una línea de detalles en el servidor tiene que existir primeramente la cabecera del pedido, debido a las restricciones de integridad referencial. Por lo tanto, la porción de la caché que contiene esta cabecera debe ser grabada antes que la porción correspondiente a las líneas de detalles.

Combinando la caché con grabaciones directas

Resumiendo lo que hemos explicado a lo largo de varios capítulos, tenemos en definitiva tres técnicas para garantizar la atomicidad de las actualizaciones de objetos complejos:

- La edición en memoria de los datos del objeto, y su posterior grabación durante una transacción.
- La edición con controles *data-aware*, activando una transacción al comenzar la operación.
- El uso de actualizaciones en caché.

De entrada, trataremos de descartar la primera técnica siempre que sea posible. La segunda técnica nos permite evitar la duplicación de verificaciones, y nos libera del problema de la transferencia de datos a las tablas. Pero no es aconsejable usar transacciones durante la edición, pues los bloqueos impuestos se mantienen todo el tiempo que el usuario necesita para la edición. Las actualizaciones en caché evitan el problema de los bloqueos mantenidos durante largos intervalos, además de que el usuario tampoco tiene que duplicar restricciones. El uso automático de transacciones durante la grabación de la caché nos asegura la atomicidad de las operaciones. Pero el comportamiento exageradamente “optimista” de las actualizaciones en caché puede plantearle dificultades al usuario y al programador.

Utilizaré como ejemplo el sistema de entrada de pedidos. Supongamos que las tablas de pedidos y las líneas de detalles tienen la caché activa. Como estas tablas se utilizan para altas, no hay problemas con el acceso concurrente a los registros generados por la aplicación. Ahora bien, si nuestra aplicación debe modificar la tabla de inventario, *tb.Articulos*, al realizar altas, ya nos encontramos en un aprieto. Si utilizamos actualizaciones en caché, no podremos vender Coca-Cola simultáneamente desde dos terminales, pues estas dos terminales intentarán modificar el mismo registro con dos versiones diferentes durante la grabación de la caché, generando un error de bloqueo optimista. En cambio, si no utilizamos caché, no podemos garantizar la atomicidad de la grabación: si, en la medida que vamos introduciendo y modificando líneas de detalles, actualizamos el inventario, estas grabaciones tienen carácter *definitivo*. La solución completa tampoco es realizar estas modificaciones en una transacción *aparte*,

posterior a la llamada a *ApplyUpdates*, pues al ser transacciones independientes no se garantiza el éxito o fracaso simultáneo de ambas.

A estas alturas, el lector se habrá dado cuenta de que, si estamos trabajando con un sistema SQL, la solución es muy fácil: implementar los cambios en la tabla de artículos en un *trigger*. De este modo, la modificación de esta tabla ocurre al transferirse el contenido de la caché a la base de datos, durante la misma transacción. Así, todas la grabaciones tienen éxito, o ninguna.

¿Y qué pasa con los pobres mortales que están obligados a seguir trabajando con Paradox, dBase, FoxPro y demás engendros locales? Para estos casos, necesitamos ampliar el algoritmo de escritura de la caché, de manera que se puedan realizar grabaciones directas durante la transacción que inicia *ApplyUpdates*. Delphi no ofrece soporte directo para esto, pero es fácil crear una función que sustituya a *ApplyUpdates*, aplicada a una base de datos. Y una de las formas de implementar estas extensiones es utilizando punteros a métodos:

```
void __fastcall ApplyUpdatesEx(
    TDBDataSet* const* DataSets, int DataSets_size,
    TNotifyEvent BeforeApply,
    TNotifyEvent BeforeCommit,
    TNotifyEvent AfterCommit)
{
    TDatabase *DB = DataSets[0]->Database;
    DB->StartTransaction();
    try
    {
        if (BeforeApply) BeforeApply(DB);
        for (int i = 0; i <= DataSets_size; i++)
            DataSets[i]->ApplyUpdates();
        if (BeforeCommit) BeforeCommit(DB);
        DB->Commit();
    }
    catch(Exception&)
    {
        DB->Rollback();
        throw;
    }
    for (int i = 0; i <= DataSets_size; i++)
        DataSets[i]->CommitUpdates();
    if (AfterCommit) AfterCommit(DB);
}
```

Los punteros a métodos *BeforeApply*, *BeforeCommit* y *AfterCommit* se han declarado como pertenecientes al tipo *TNotifyEvent* por comodidad y conveniencia. Observe que cualquier escritura que se realice en los métodos *BeforeApply* y *BeforeCommit* se realiza dentro de la misma transacción en la que se graba la caché. La llamada a *AfterCommit*, en cambio, se realiza después del bucle en que se vacía la caché, utilizando el método *CommitUpdates*. Esto lo hemos programado así para evitar que una excepción producida en la llamada a este evento impida el vaciado de la caché.

He marcado la diferencia entre *BeforeApply* y *BeforeCommit* por causa de un comportamiento algo anómalo de las actualizaciones en caché: cuando se ha llamado a *ApplyUpdates*, pero todavía no se ha vaciado la caché de la tabla con *CommitUpdates*, los registros modificados aparecen dos veces dentro del cursor de la tabla. Si necesitamos un método que recorra alguna de las tablas actualizadas, como el que veremos dentro de poco, debemos conectarlo a *BeforeApply* mejor que a *BeforeCommit*.

Si estamos utilizando el evento *OnCloseQuery* de la ficha para automatizar la grabación y cancelación de cambios, tenemos que extender la función *PuedoCerrar* para que acepte los punteros a métodos como parámetros. He incluido también la posibilidad de realizar la entrada de datos en modo continuo, como se ha explicado en el capítulo sobre actualizaciones. Esta es la nueva versión:

```

bool __fastcall PuedoCerrar(TForm *AForm,
    TDBDataSet* const* DataSets, int DataSets_size,
    bool ModoContinuo,
    TNotifyEvent BeforeApply,
    TNotifyEvent BeforeCommit,
    TNotifyEvent AfterCommit)
{
    // Verificamos si hay cambios en caché
    bool Actualizar = False;
    for (int i = 0; i <= DataSets_size; i++)
        if (DataSets[i]->UpdatesPending || DataSets[i]->Modified)
            Actualizar = True;
    // Nos han pedido que grabemos los cambios
    if (AForm->ModalResult == mrOk)
    {
        ApplyUpdatesEx(DataSets, DataSets_size,
            BeforeApply, BeforeCommit, AfterCommit);
        if (ModoContinuo)
        {
            DataSets[0]->Append();
            return False;
        }
    }
    // Hay que deshacer los cambios
    else if (! Actualizar ||
        Application->MessageBox("¿Desea abandonar los cambios?",
            "Atención", MB_ICONQUESTION | MB_YESNO) == IDYES)
        for (int i = 0; i <= DataSets_size; i++)
            DataSets[i]->CancelUpdates();
    // El usuario se arrepiente de abandonar los cambios
    else
        return False;
    return True;
}

```

Para implementar la entrada de datos continua se ha supuesto que la primera tabla del vector es la tabla principal, a partir de la cual se desarrolla todo el proceso de actualización.

Ahora volvemos al ejemplo que motivó estas extensiones. Supongamos que tenemos una ficha de entrada de pedidos y queremos actualizar el inventario, en la tabla *tbArticulos*, una vez grabado el pedido. La solución consiste en declarar un método público en el módulo de datos que realice los cambios en la tabla de artículos de acuerdo a las líneas de detalles del pedido activo:

```
void __fastcall TmodDatos::ActualizarInventario(TObject *Sender)
{
    TLocateOptions Opt;
    tbDetalles->First();
    while (! tbDetalles->Eof)
    {
        if (tbArticulos->Locate("PartNo",
            tbDetalles->FieldValues["PartNo"], Opt)
        {
            tbArticulos->Edit();
            tbArticulos->FieldValues["OnOrder"] =
                tbArticulos->FieldValues["OnOrder"] +
                tbDetalles->FieldValues["Qty"];
            tbArticulos->Post();
        }
        tbDetalles->Next();
    }
}
```

Luego, en la ficha de altas de pedidos modificamos la respuesta al evento *OnCloseQuery* de esta forma:

```
void __fastcall entPedidos::entPedidosCloseQuery(TObject *Sender,
    bool &CanClose)
{
    CanClose = PuedoCerrar(this, OPENARRAY(TBDataSet*,
        (modDatos->tbPedidos, modDatos->tbDetalles)),
        True, ActualizarInventario, NULL, NULL);
}
```

Hemos especificado el puntero nulo *NULL* como valor para los parámetros *BeforeCommit* y *AfterCommit*. Este último parámetro puede aprovecharse, por ejemplo, para realizar la impresión de los datos del pedido una vez que se ha confirmado su entrada.

Los sistemas cliente/servidor también pueden obtener beneficios de esta técnica. En el mismo ejemplo de altas de pedidos podemos utilizar para actualizar el inventario un procedimiento almacenado que se ejecutaría después de haber grabado la caché, en vez de apoyarnos en el uso de *triggers*. Una ventaja adicional es que minimizamos la posibilidad de un *deadlock*, o abrazo mortal.

Prototipos y métodos virtuales

Realmente, el uso de punteros a métodos es cuando menos engorroso, pero es necesario mientras *PuedoCerrar* sea una función, no un método. Una forma de simplificar esta metodología de trabajo es definiendo un prototipo o plantilla de formulario de entrada de datos, que se utilice como clase base para todas las altas y modificaciones de objetos complejos. En este caso, *PuedoCerrar* puede definirse como un método de esta clase, y los punteros a métodos sustituirse por métodos virtuales.

Partamos de un nuevo formulario, al cual configuraremos visualmente como un cuadro de diálogo, y al cual añadiremos los siguientes métodos en su declaración de clase:

```
class TwndDialogo : public TForm
{
    // ...
protected:
    virtual void BeforeApply(TDatabase *DB);
    virtual void BeforeCommit(TDatabase *DB);
    virtual void AfterCommit(TDatabase *DB);
    void ApplyUpdatesEx(TDBDataSet* const* DataSets,
        int DataSets_size);
    bool PuedoCerrar(TDBDataSet* const* DataSets,
        int DataSets_size, bool ModoContinuo);
    // ...
};
```

En principio, los tres métodos virtuales se definen con cuerpos vacíos:

```
void TwndDialogo::BeforeApply(TDatabase *DB)
{
}

void TwndDialogo::BeforeCommit(TDatabase *DB)
{
}

void TwndDialogo::AfterCommit(TDatabase *DB)
{
}
```

La nueva implementación de *ApplyUpdatesEx* ejecuta estos métodos:

```
void __fastcall TwndDialogo::ApplyUpdatesEx(
    TDBDataSet* const* DataSets, int DataSets_size)
{
    TDatabase *DB = DataSets[0]->Database;
    DB->StartTransaction();
    try
    {
        BeforeApply(DB);
        for (int i = 0; i <= DataSets_size; i++)
            DataSets[i]->ApplyUpdates();
    }
}
```

```

        BeforeCommit(DB);
        DB->Commit();
    }
    catch(Exception&)
    {
        DB->Rollback();
        throw;
    }
    for (int i = 0; i <= DataSets_size; i++)
        DataSets[i]->CommitUpdates();
    AfterCommit(DB);
}

```

Finalmente, *PuedoCerrar* se simplifica bastante:

```

bool __fastcall TwndDialogo::PuedoCerrar(
    TDBDataSet* const* DataSets, int DataSets_size,
    bool ModoContinuo)
{
    // Verificamos si hay cambios en caché
    int i = DataSets_size;
    while (i >= 0 && ! DataSets[i]->UpdatesPending &&
        ! DataSets[i]->Modified) i--;
    // Nos han pedido que grabemos los cambios
    if (ModalResult == mrOK)
    {
        ApplyUpdatesEx(DataSets, DataSets_size);
        if (ModoContinuo)
        {
            DataSets[0]->Append();
            return False;
        }
    }
    // Hay que deshacer los cambios
    else if (i < 0 || // ¿Hay cambios?
        Application->MessageBox("¿Desea abandonar los cambios?",
            "Atención", MB_ICONQUESTION | MB_YESNO) == IDYES)
        for (i = 0; i <= DataSets_size; i++)
            DataSets[i]->CancelUpdates();
    // El usuario se arrepiente de abandonar los cambios
    else
        return False;
    return True;
}

```

Partiendo de este prototipo, el programador puede derivar de él los formularios de entrada de datos, redefiniendo los métodos virtuales sólo cuando sea necesario.

Cómo actualizar consultas “no” actualizables

Cuando una tabla o consulta ha activado las actualizaciones en caché, la grabación de los cambios almacenados en la memoria caché es responsabilidad del BDE. Normalmente, el algoritmo de actualización es generado automáticamente por el BDE, pero tenemos también la posibilidad de especificar la forma en que las actualizaciones

tienen lugar. Antes, en el capítulo 26, hemos visto cómo podíamos modificar el algoritmo de grabación mediante la propiedad *UpdateMode* de una tabla. Bien, ahora podemos ir más lejos, y no solamente con las tablas.

Esto es especialmente útil cuando estamos trabajando con consultas contra un servidor SQL. Las bases de datos SQL se ajustan casi todas al estándar del 92, en el cual se especifica que una sentencia **select** no es actualizable cuando contiene un encuentro entre tablas, una cláusula **distinct** o **group by**, etc. Por ejemplo, InterBase no permite actualizar la siguiente consulta, que muestra las distintas ciudades en las que viven nuestros clientes:

```
select distinct City
from Customer
```

No obstante, es fácil diseñar reglas para la actualización de esta consulta. La más sencilla es la relacionada con la modificación del nombre de una ciudad. En tal caso, deberíamos modificar ese nombre en todos los registros de clientes que lo mencionan. Más polémica es la interpretación de un borrado. Podemos eliminar a todos los clientes de esa ciudad, tras pedirle confirmación al usuario. En cuanto a las inserciones, lo más sensato es prohibirlas, para no vernos en la situación de Alicia, que había visto muchos gatos sin sonrisa, pero nunca una sonrisa sin gato.

C++ Builder nos permite intervenir en el mecanismo de actualización de conjuntos de datos en caché al ofrecernos el evento *OnUpdateRecord*:

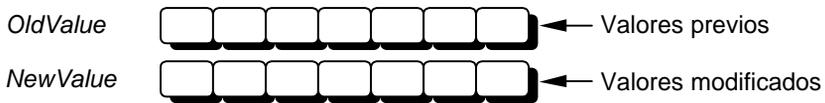
```
typedef void __fastcall (__closure *TUpdateRecordEvent)
(TDataSet* DataSet, TUpdateKind UpdateKind,
TUpdateAction &UpdateAction);
```

El parámetro *DataSet* representa al conjunto de datos que se está actualizando. El segundo parámetro puede tener uno de estos valores: *ukInsert*, *ukModify* ó *ukDelete*, para indicar qué operación se va a efectuar con el registro activo de *DataSet*. El último parámetro debe ser modificado para indicar el resultado del evento. He aquí los posibles valores:

Valor	Significado
<i>uaFail</i>	Anula la aplicación de las actualizaciones, lanzando una excepción
<i>uaAbort</i>	Aborta la operación mediante la excepción silenciosa
<i>uaSkip</i>	Ignora este registro en particular
<i>uaRetry</i>	No se utiliza en este evento
<i>uaApplied</i>	Actualización exitosa

Dentro de la respuesta a este evento, podemos combinar cuantas operaciones de actualización necesitemos, siempre que no cambiemos la fila activa de la tabla que se actualiza. Cuando estamos trabajando con este evento, tenemos acceso a un par de

propiedades especiales de los campos: *OldValue* y *NewValue*, que representan el valor del campo antes y después de la operación, al estilo de las variables *new* y *old* de los *triggers* en SQL.



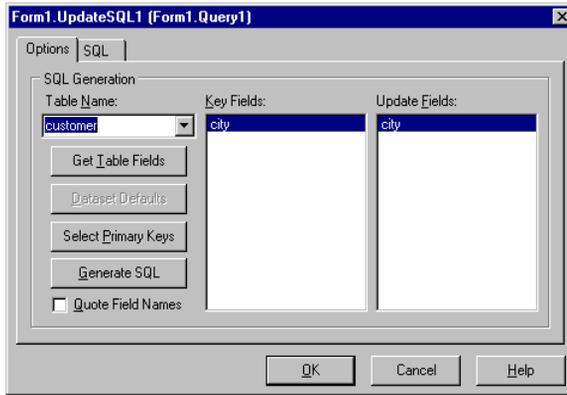
Sin embargo, lo más frecuente es que cada tipo de actualización pueda realizarse mediante una sola sentencia SQL. Para este caso sencillo, las tablas y consultas han previsto una propiedad *UpdateObject*, en la cual se puede asignar un objeto del tipo *TUpdateSQL*. Este componente actúa como depósito para tres instrucciones SQL, almacenadas en las tres propiedades *InsertSQL*, *ModifySQL* y *DeleteSQL*. Primero hay que asignar este objeto a la propiedad *UpdateObject* de la consulta. La operación tiene un efecto secundario inverso: asignar la consulta a la propiedad no publicada *DataSet* del componente *TUpdateSQL*. Sin esta asignación, no funciona ni la generación automática de instrucciones que vamos a ver ahora, ni la posterior sustitución de parámetros en tiempo de ejecución.

Una vez que el componente *TUpdateSQL* está asociado a un conjunto de datos, podemos hacer doble clic sobre él para ejecutar un editor de propiedades, que nos ayudará a generar el código de las tres sentencias SQL. El editor del componente *TUpdateSQL* necesita que le especifiquemos cuál es la tabla que queremos actualizar, pues en el caso de un encuentro tendríamos varias posibilidades. En nuestro caso, se trata de una consulta muy simple, por lo que directamente pulsamos el botón *Generate SQL*, y nos vamos a la página siguiente, para retocar las instrucciones generadas si es necesario. Las instrucciones generadas son las siguientes:

```
// Borrado
delete from Customer
where City = :OLD_City

// Modificación
update Customer
set City = :City
where City = :OLD_City

// Altas
insert into Customer(City)
values (:City)
```



Las instrucciones generadas utilizan parámetros especiales, con el prefijo *OLD_*, de modo similar a la variable de contexto *old* de los *triggers* de InterBase y Oracle. Es evidente que la instrucción **insert** va a producir un error, al no suministrar valores para los campos no nulos de la tabla. Pero recuerde que de todos modos no tenía sentido realizar inserciones en la consulta, y que no íbamos a permitirlo.

El evento *OnUpdateRecord*

Si un conjunto de datos con actualizaciones en caché tiene un objeto enganchado en su propiedad *UpdateObject*, y no se ha definido un manejador para *OnUpdateRecord*, el conjunto de datos utiliza directamente las instrucciones SQL del objeto de actualización para grabar el contenido de su caché. Pero si hemos asignado un manejador para el evento mencionado, se ignora el objeto de actualización, y todas las grabaciones deben efectuarse en la respuesta al evento. Para restaurar el comportamiento original, debemos utilizar el siguiente código:

```
void __fastcall TmodDatos::Query1UpdateRecord(TDataSet *DataSet,
    TUpdateKind UpdateKind, TUpdateAction &UpdateAction)
{
    UpdateSQL1->DataSet = Query1;
    UpdateSQL1->Apply(UpdateKind);
    // O, alternativamente:
    // UpdateSQL1->SetParams(UpdateKind);
    // UpdateSQL1->ExecSql(UpdateKind);
    UpdateAction = uaApplied;
}
```

En realidad, la primera asignación del método anterior sobra, pero es indispensable si el objeto de actualización no está asociado directamente a un conjunto de datos mediante la propiedad *UpdateObject*, sino que, por el contrario, es independiente. Como observamos, el método *Apply* realiza primero la sustitución de parámetros y luego ejecuta la sentencia que corresponde. La sustitución de parámetros es especial, por-

que debe tener en cuenta los prefijos *OLD_*. En cuanto al método *ExecSQL*, está programado del siguiente modo (en Delphi, naturalmente):

```

procedure TUpdateSQL.ExecSQL(UpdateKind: TUpdateKind);
begin
  with Query[UpdateKind] do
    begin
      Prepare;
      ExecSQL;
      if RowsAffected <> 1 then DatabaseError(SUpdateFailed);
    end;
end;

```

¿Se da cuenta de que no podemos utilizar el comportamiento por omisión para nuestra consulta, aparentemente tan simple? El problema es que *ExecSQL* espera que su ejecución afecte exactamente a una fila, mientras que nosotros queremos que al modificar un nombre de ciudad se puedan modificar potencialmente varios usuarios. No tendremos más solución que programar nuestra propia respuesta al evento:

```

void __fastcall TmodDatos::Query1UpdateRecord(TDataSet *DataSet,
  TUpdateKind UpdateKind, TUpdateAction &UpdateAction)
{
  if (UpdateKind == ukInsert)
    DatabaseError("No se permiten inserciones", 0);
  else if (UpdateKind == ukDelete &&
    MessageDlg("¿Está seguro?", mtConfirmation,
      TMsgDlgButtons() << mbYes << mbNo, 0) != mrYes)
  {
    UpdateAction = uaAbort;
    return;
  }
  UpdateSQL1->SetParams(UpdateKind);
  UpdateSQL1->Query[UpdateKind]->ExecSQL();
  UpdateAction = uaApplied;
}

```

En algunas ocasiones, la respuesta al evento *OnUpdateRecord* se basa en ejecutar procedimientos almacenados, sobre todo cuando las actualizaciones implican modificar simultáneamente varias tablas.

Detección de errores durante la grabación

Cuando se producen errores durante la confirmación de las actualizaciones en caché, éstos no se reportan por medio de los eventos ya conocidos, *OnPostError* y *OnDeleteError*, sino a través del nuevo evento *OnUpdateError*:

```

void __fastcall TDataModule1::Table1UpdateError(TDataSet *DataSet,
  EDatabaseError* E, TUpdateKind UpdateKind,
  TUpdateAction &UpdateAction);

```

La explicación es sencilla: como la grabación del registro se difiere, no es posible verificar violaciones de claves primarias, de rangos, y demás hasta que no se apliquen las actualizaciones.

Como se puede ver, la filosofía de manejo de este evento es similar a la del resto de los eventos de aviso de errores. Se nos indica el conjunto de datos, *DataSet* y la excepción que está a punto de producirse. Se nos indica además qué tipo de actualización se estaba llevando a cabo: si era una modificación, una inserción o un borrado (*UpdateKind*). Por último, mediante el parámetro *UpdateAction* podemos controlar el comportamiento de la operación. Los valores son los mismos que para el evento *OnUpdateRecord*, pero varía la interpretación de dos de ellos:

Valor	Nuevo significado
<i>uaRetry</i>	Reintenta la operación sobre este registro
<i>uaApplied</i>	¡No lo utilice en este evento!

A diferencia de lo que sucede con los otros eventos de errores, en este evento tenemos la posibilidad de ignorar el error o de corregir nosotros mismos el problema realizando la actualización. La mayor parte de los ejemplos de utilización de este evento tratan los casos más sencillos: ha ocurrido alguna violación de integridad, la cual es corregida y se reintenta la operación. Sin embargo, el caso más frecuente de error de actualización en la práctica es el fallo de un bloqueo optimista: alguien ha cambiado el registro original mientras nosotros, reflexionando melancólicamente como Hamlet, decidíamos si aplicar o no las actualizaciones. El problema es que, en esta situación, no tenemos forma de restaurar el registro *antiguo* para reintentar la operación. Recuerde que, como explicamos al analizar la propiedad *UpdateMode*, el BDE realiza la actualización de un registro de una tabla SQL mediante una instrucción SQL en la que se hace referencia al registro mediante sus valores anteriores. Para generar la sentencia de actualización, se utilizan los valores almacenados en los campos, en sus propiedades *OldValue*. Estas propiedades son sólo para lectura de modo que, una vez que nos han cambiado el registro original, no tenemos forma de saber en qué lo han transformado.

Sin embargo, en casos especiales, este problema tiene solución. Supongamos que estamos trabajando con la tabla *parts*, el inventario de productos. El conflicto que surge con mayor frecuencia es que alguien vende 18 Coca-Colas, a la par que nosotros intentamos vender otras 15. Las cantidades vendidas se almacenan en la columna *OnOrder*. Evidentemente, se producirá un error si el segundo usuario realiza su transacción después que hemos leído el registro de las Coca-Colas, pero antes de que hayamos grabado nuestra operación. El problema se produce por un pequeño desliz de concepto: a mí no me interesa decir que se han vendido $100+15=115$ unidades, sino que se han vendido otras 15. Esta operación puede llevarse a cabo fácilmente mediante una consulta paralela, *Query1*, cuya instrucción SQL sea la siguiente:

```

update Parts
set    OnOrder = OnOrder + :NewOnOrder - :OldOnOrder
where  PartNo = :PartNo

```

En estas circunstancias, la respuesta al evento *OnUpdateError* de la tabla de artículos puede ser la siguiente:

```

void __fastcall TForm1::TableUpdateError(TDataSet *TDataSet,
    EDatabaseError *E, TUpdateKind *UpdateKind,
    TUpdateAction &UpdateAction)
{
    EDBEngineError *Err = dynamic_cast<EDBEngineError*>(E);
    if (UpdateKind == ukModify && Err)
        if (Err->Errors[0]->ErrorCode == DBIERR_OPTRELOCKFAILED)
        {
            Query1->ParamByName("PartNo")->AsFloat =
                Table1->FieldByName("PartNo")->OldValue;
            Query1->ParamByName("OldOnOrder")->AsInteger =
                Table1->FieldByName("OnOrder")->OldValue;
            Query1->ParamByName("NewOnOrder")->AsInteger =
                Table1->FieldByName("OnOrder")->NewValue;
            Query1->ExecSQL();
            UpdateAction = uaSkip;
        }
    }
}

```

Hemos tenido cuidado en aplicar la modificación únicamente cuando el error es el fallo de un bloqueo optimista; podíamos haber sido más cuidadosos, comprobando que la única diferencia entre las versiones fuera el valor almacenado en la columna *OnOrder*. Un último consejo: después de aplicar los cambios es necesario vaciar la caché, pues los registros que provocaron conflictos han quedado en la misma, gracias a la asignación de *uaSkip* al parámetro *Update.Action*.

¿Tablas ... o consultas en caché?

¿Recuerda que, cuando a una consulta se le asigna en *RequestLive* el valor *True*, el BDE genera instrucciones de recuperación sobre las tablas de esquema de la base de datos? Estas instrucciones que se lanzan en segundo plano al servidor durante la apertura de la consulta son idénticas a las que el BDE siempre lanza con las tablas, y hacen que la apertura de la consulta tarde bastante.

Cuando la consulta se abre con actualizaciones en caché, y se utiliza alguno de los métodos anteriores para indicar manualmente las instrucciones de actualización, el BDE no necesita ejecutar el protocolo de apertura antes mencionado. De este modo, la apertura de la consulta es casi instantánea (en realidad, dependerá del tiempo que tarde el servidor en evaluarla, pero si se trata de una consulta “con sentido”, el tiempo necesario será mínimo gracias a la optimización).

Otro problema grave de las consultas se resuelve también con esta técnica. Recordará que cuando insertábamos un registro en una consulta actualizable, había que cerrarla y volverla a abrir para que el nuevo registro apareciera en el cursor. Pero cuando se utilizan actualizaciones en caché no es necesario refrescar el cursor para que aparezca la nueva fila.

¿Quiere decir todo esto que la mejor opción para navegación y mantenimiento son las consultas con actualizaciones en caché? La respuesta es un *sí*, pero con los siguientes reparos:

1. Al utilizar actualizaciones en caché se le complicará algo la interfaz de usuario. El usuario medio espera que, al cambiar de fila activa, las modificaciones que haya realizado en la misma se graben inmediatamente.
2. Las actualizaciones en caché no resuelven los problemas asociados a la navegación sobre conjuntos de datos grandes. Si usted puede siempre limitar inteligentemente el tamaño de los conjuntos de datos a navegar, no hay problema alguno. En caso contrario, las tablas siguen siendo una buena idea.

Tenga en cuenta que, debido a lo complicado de la implementación de las actualizaciones en caché, siguen existiendo *bugs* en el BDE, incluso en la versión 5.0.1 que estamos utilizando durante la redacción de este libro.

4

Programación distribuida

- Conjuntos de datos clientes
- El Modelo de Objetos Componentes
- Servidores COM
- Automatización OLE: controladores
- Automatización OLE: servidores
- Midas
- Servidores de Internet

Parte

Conjuntos de datos clientes

SER ORGANIZADOS TIENE SU RECOMPENSA, Y NO HAY que esperar a morirse e ir al Cielo para disfrutar de ella. Al parecer, los programadores de Borland (perdón, Inprise) hicieron sus deberes sobresalientemente al diseñar la biblioteca dinámica *dbclient.dll*, y el tipo de datos *TClientDataSet*, que se comunica con ella. Al aislar el código de manejo de bases de datos en memoria en esta biblioteca, han podido mejorar ostensiblemente las posibilidades de la misma al desarrollar la versión 4 de la VCL.

Los *conjuntos de datos clientes*, o *client datasets*, son objetos que pertenecen a una clase derivada de *TDataSet*: la clase *TClientDataSet*. Estos objetos almacenan sus datos en memoria RAM local, pero la característica que los hizo famosos en C++ Builder 3 es que pueden leer estos datos desde un servidor de datos remoto mediante automatización DCOM. En la edición anterior de este libro, los conjuntos de datos clientes se explicaban junto a las técnicas de comunicación con bases de datos remotas mediante Midas. Pero, como he explicado antes, hay tantas nuevas características que estos componentes se han ganado un capítulo independiente.

Aquí nos concentraremos en las características de *TClientDataSet* que nos permiten gestionar bases de datos en memoria, utilizando como origen de datos a ficheros “planos” del sistema operativo. ¿Qué haría usted si tuviera que implementar una aplicación de bases de datos, pero cuyo volumen de información esté en el orden de 1.000 a 10.000 registros? En vez de utilizar Paradox, dBase o Access, que requieren la presencia de un voluminoso y complicado motor de datos, puede elegir los conjuntos de datos en memoria. Veremos que en C++ Builder 4 estos conjuntos soportan características avanzadas como tablas anidadas, valores agregados mantenidos, índices dinámicos, etc. Para más adelante dejaremos las propiedades, métodos y eventos que hacen posible la comunicación con servidores Midas.

Creación de conjuntos de datos

El componente *TClientDataSet* está situado en la página *Midas* de la Paleta de Componentes. Usted trae uno de ellos a un formulario y se pregunta: ¿cuál es el esquema

de los datos almacenados por el componente? Si el origen de nuestros datos es un servidor remoto, la definición de campos, índices y restricciones del conjunto de datos se leen desde el servidor, por lo cual no tenemos que preocuparnos en ese sentido.

Supongamos ahora que los datos deban extraerse de un fichero local. Este fichero debe haber sido creado por otro conjunto de datos *TClientDataSet*, por lo cual en algún momento tenemos que definir el esquema del conjunto y crearlo, para evitar un círculo vicioso. Para crear un conjunto de datos en memoria debemos definir los campos e índices que deseemos, y aplicar posteriormente el método *CreateDataSet*:

```
TFieldDefs *fd = ClientDataSet1->FieldDefs;
fd->Clear();
fd->Add("Nombre", ftString, 30, True);
fd->Add("Apellidos", ftString, 30, True);
fd->Add("Direccion", ftString, 30, True);
fd->Add("Telefono", ftString, 10, True);
fd->Add("FAX", ftString, 10, True);
fd->Add("Mail", ftString, 80, True);
ClientDataSet1->CreateDataSet();
```

Al igual que sucede con las tablas, en C++ Builder 4 existe la alternativa de llenar la propiedad *FieldDefs* en tiempo de diseño, con lo cual la propiedad *StoredDefs* se vuelve verdadera. Cuando deseemos crear el conjunto de datos en tiempo de ejecución, solamente tenemos que llamar a *CreateDataSet*. Si no existen definiciones en *FieldDefs*, pero sí se han definido campos persistentes, se utilizan estos últimos para crear el conjunto de datos.

También se puede crear el conjunto de datos junto a sus índices, utilizando la propiedad *IndexDefs* en tiempo de diseño o de ejecución:

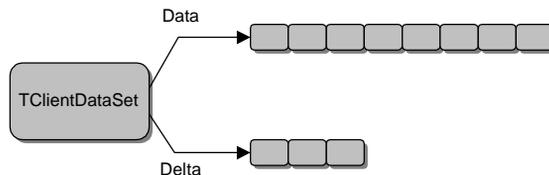
```
// Definir campos
TFieldDefs *fd = ClientDataSet1->FieldDefs;
fd->Clear();
fd->Add("Nombre", ftString, 30, True);
fd->Add("Apellidos", ftString, 30, True);
fd->Add("Direccion", ftString, 30, False);
fd->Add("Telefono", ftInteger, 0, True);
fd->Add("FAX", ftInteger, 0, False);
fd->Add("Mail", ftString, 80, False);
// Definir índices
TIndexDefs *id = ClientDataSet1->IndexDefs;
id->Clear();
id->Add("xNombre", "Nombre",
    TIndexOptions() << ixCaseInsensitive);
id->Add("xApellidos", "Apellidos",
    TIndexOptions() << ixCaseInsensitive);
id->Add("xTelefono", "Telefono", TIndexOptions());
// Crear el conjunto de datos
ClientDataSet1->CreateDataSet();
```

Más adelante veremos todos los tipos de índices que admiten los conjuntos de datos clientes.

Por último, también podemos crear el conjunto de datos cliente en tiempo de diseño, pulsando el botón derecho del ratón sobre el componente y ejecutando el comando del menú local *Create DataSet*.

Cómo el *TClientDataSet* obtiene sus datos

Un componente *TClientDataSet* siempre almacena sus datos en dos vectores situados en la memoria del ordenador donde se ejecuta la aplicación. Al primero de estos vectores se accede mediante la propiedad *Data*, de tipo *OleVariant*, y contiene los datos leídos inicialmente por el componente; después veremos desde dónde se leen estos datos. La segunda propiedad, del mismo tipo que la anterior, se denomina *Delta*, y contiene los cambios realizados sobre los datos iniciales. Los datos reales de un *TClientDataSet* son el resultado de mezclar el contenido de las propiedades *Data* y *Delta*.



Como mencionaba en la introducción, hay varias formas de obtener estos datos:

- A partir de ficheros del sistema operativo.
- Copiando los datos almacenados en otro conjunto de datos.
- Mediante una conexión a una interfaz *IProvider* suministrada por un servidor de datos remoto.

Para obtener datos a partir de un fichero, se utiliza el método *LoadFromFile*. El fichero debe haber sido creado por un conjunto de datos (quizás éste mismo) mediante una llamada al método *SaveToFile*, que también está disponible en el menú local del componente. Los ficheros, cuya extensión por omisión es *cds*, no almacenan los registros en formato texto, sino en un formato que es propio de estos componentes. Podemos leer un conjunto de datos cliente a partir de un fichero aunque el componente no contenga definiciones de campos, ya sea en *FieldDefs* o en *Fields*.

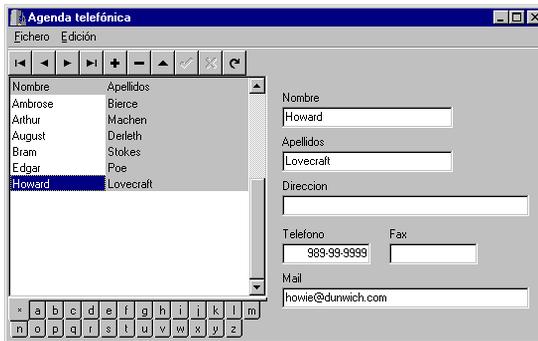
Sin embargo, la forma más común de trabajar con ficheros es asignar un nombre de fichero en la propiedad *FileName* del componente. Si el fichero existe en el momento

de la apertura del conjunto de datos, se lee su contenido. En el momento en que se cierra el conjunto de datos, se sobrescribe el contenido del fichero con los datos actuales.

Navegación, búsqueda y selección

Como todo buen conjunto de datos, los *TClientDataSet* permiten las ya conocidas operaciones de navegación y búsqueda: *First*, *Next*, *Last*, *Prior*, *Locate*, rangos, filtros, etc. Permiten también especificar un criterio de ordenación mediante las propiedades *IndexName* e *IndexFieldNames*. En esta última propiedad, al igual que sucede con las tablas SQL, podemos indicar cualquier combinación de columnas, pues el conjunto de datos cliente puede generar dinámicamente un índice de ser necesario.

La siguiente imagen muestra una sencilla aplicación para el mantenimiento de una libreta de teléfonos. La aplicación completa viene incluida en el CD-ROM del libro:



En todo momento, el conjunto de datos se encuentra ordenado por el campo *Nombre* o por el campo *Apellidos*. El orden se puede cambiar pulsando el ratón sobre el título de la columna en la rejilla:

```
void __fastcall TwndMain::DBGrid1TitleClick(TColumn *Column)
{
    ClientDataSet1->CancelRange();
    ClientDataSet1->IndexFieldNames = Column->FieldName;
    TabControl1->TabIndex = 0;
}
```

Debajo de la rejilla he situado un componente *TTabControl*, con todas las letras del alfabeto. En respuesta al evento *OnChange* de este control, se ejecuta este método:

```
void __fastcall TwndMain::TabControl1Change(TObject *Sender)
{
    AnsiString Tab;
    Tab = TabControl1->Tabs->Strings[TabControl1->TabIndex];
```

```

if (Tab == "")
    ClientDataSet1->CancelRange();
else
    ClientDataSet1->SetRange(ARRAYOFCONST((Tab)),
        ARRAYOFCONST((Tab + "zzz")));
}

```

Por supuesto, los índices que se han definido sobre este conjunto de datos no hacen distinción entre mayúsculas y minúsculas.

Hay una simpática peculiaridad de estos conjuntos de datos que concierne a la navegación: la propiedad *RecNo* no solamente nos indica en qué número de registro estamos situados, sino que además permite que nos movamos a un registro determinado asignando su número de posición en la misma:

```

void __fastcall TwndMain::IrAExecute(TObject *Sender)
{
    AnsiString S;
    if (InputQuery("Ir a posición", "Número de registro:", S))
        ClientDataSet1->RecNo = StrToInt(S);
}

```

Filtros

Una agradable sorpresa: las expresiones de filtros de los conjuntos de datos clientes soportan montones de características no permitidas por los conjuntos de datos del BDE. En primer lugar, ya podemos saber si un campo es nulo o no utilizando la misma sintaxis que en SQL:

```
Direccion is null
```

Podemos utilizar expresiones aritméticas:

```
Descuento * Cantidad * Precio < 100
```

Se han añadido las siguientes funciones de cadenas:

```
upper, lower, substring, trim, trimleft, trimright
```

Y disponemos de estas funciones para trabajar con fechas:

```
day, month, year, hour, minute, second, date, time, getdate
```

La función *date* extrae la parte de la fecha de un campo de fecha y hora, mientras que *time* aísla la parte de la hora. La función *getdate* devuelve el momento actual. También se permite la aritmética de fechas:

```
getdate - HireDate > 365
```

Por último, podemos utilizar los operadores **in** y **like** de SQL:

```
Nombre like '%soft'
year(FechaVenta) in (1994,1995)
```

Edición de datos

Como hemos explicado, los datos originales de un *TClientDataSet* se almacenan en su propiedad *Data*, mientras que los cambios realizados van a parar a la propiedad *Delta*. Cada vez que un registro se modifica, se guardan los cambios en esta última propiedad, pero sin descartar o mezclar con posibles cambios anteriores. Esto permite que el programador ofrezca al usuario comandos sofisticados para deshacer los cambios realizados, en comparación con las posibilidades que ofrece un conjunto de datos del BDE.

En primer lugar, tenemos el método *UndoLastChange*:

```
void __fastcall TClientDataSet::UndoLastChange(bool FollowChanges);
```

Este método deshace el último cambio realizado sobre el conjunto de datos. Cuando indicamos *True* en su parámetro, el cursor del conjunto de datos se desplaza a la posición donde ocurrió la acción. Con *UndoLastChange* podemos implementar fácilmente el típico comando *Deshacer* de los procesadores de texto. Claro, nos interesa también saber si hay cambios, para activar o desactivar el comando de menú, y en esto nos ayuda la siguiente propiedad:

```
__property int ChangeCount;
```

Tenemos también la propiedad *SavePoint*, de tipo entero. El valor de esta propiedad indica una posición dentro del vector de modificaciones, y sirve para establecer un punto de control dentro del mismo. Por ejemplo, supongamos que usted desea imitar una transacción sobre un conjunto de datos cliente para determinada operación larga. Esta pudiera ser una solución sencilla:

```
void __fastcall TwndMain::OperacionLarga()
{
    int SP = ClientDataSet1->SavePoint;
    try
    {
        // Aquí se realizan las distintas modificaciones ...
    }
    catch(Exception&)
    {
        ClientDataSet1->SavePoint = SP;
    }
}
```

```

        throw;
    }
}

```

A diferencia de lo que sucede con una transacción verdadera, podemos anidar varios puntos de verificación.

Cuando se guardan los datos de un *TClientDataSet* en un fichero, se almacenan por separado los valores de las propiedades *Delta* y *Data*. De este modo, cuando se reanuda una sesión de edición que ha sido guardada, el usuario sigue teniendo a su disposición todos los comandos de control de cambios. Sin embargo, puede que nos interese mezclar definitivamente el contenido de estas dos propiedades, para lo cual debemos utilizar el método siguiente:

```
void __fastcall TClientDataSet::MergeChangeLog();
```

Como resultado, el espacio necesario para almacenar el conjunto de datos disminuye, pero las modificaciones realizadas hasta el momento se hacen irreversibles. Si le interesa que todos los cambios se guarden directamente en *Data*, sin utilizar el *log*, debe asignar *False* a la propiedad *LogChanges*.

Estos otros dos métodos también permiten deshacer cambios, pero de forma radical:

```
void __fastcall TClientDataSet::CancelUpdates();
void __fastcall TClientDataSet::RevertRecord();
```

Como puede ver, estos métodos son análogos a los que se utilizan en las actualizaciones en caché. Recuerde que cancelar todos los cambios significa, en este contexto, vaciar la propiedad *Delta*, y que si usted no ha mezclado los cambios mediante *MergeChangeLog* en ningún momento, se encontrará de repente con un conjunto de datos vacío entre sus manos.

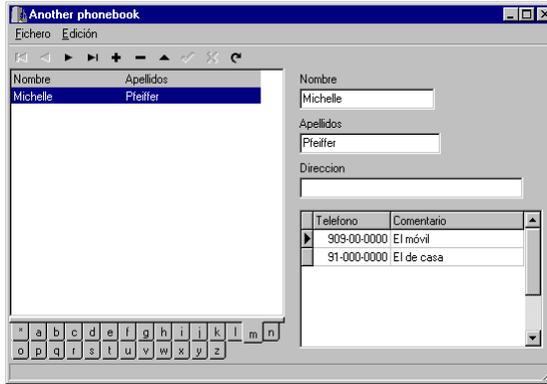
Conjuntos de datos anidados

Los conjuntos de datos clientes de la versión 4 de C++ Builder permiten definir campos de tipo *fiDataSet*, que contienen tablas anidadas. Este recurso puede servir como alternativa al uso de tablas conectadas en relaciones *master/detail*. Las principales ventajas son:

- Todos los datos se almacenan en el mismo fichero.
- Las tablas anidadas funcionan a mayor velocidad que las relaciones *master/detail*.
- Las técnicas de deshacer cambios funcionan en sincronía para los datos de cabecera y los datos anidados.

- Como las transacciones en bases de datos tradicionales pueden simularse mediante el registro de cambios, podemos realizar transacciones atómicas que incluyan a los datos de cabecera y los de detalles, simultáneamente.

La siguiente imagen muestra una variante de la aplicación de la libreta de teléfonos que utiliza una tabla anidada para representar todos los números de teléfonos que puede poseer cada persona.



Comencemos por definir el conjunto de datos. Traemos un componente de tipo *TClientDataSet* al formulario, lo nombramos *People*, vamos a su propiedad *FieldDefs* y definimos los siguientes campos:

Campo	Tipo	Tamaño	Atributos
<i>Nombre</i>	<i>ftString</i>	30	<i>ftRequired</i>
<i>Apellidos</i>	<i>ftString</i>	30	<i>ftRequired</i>
<i>Direccion</i>	<i>ftString</i>	35	
<i>Telefonos</i>	<i>ftDataSet</i>	0	

Este es el formato de los datos de cabecera. Para definir los detalles, seleccionamos la definición del campo *Telefonos*, y editamos su propiedad *ChildDefs*:

Campo	Tipo	Tamaño	Atributos
<i>Telefono</i>	<i>ftInteger</i>	0	<i>ftRequired</i>
<i>Comentario</i>	<i>ftString</i>	20	

También añadiremos un par de índices persistentes al conjunto de datos. Editamos la propiedad *IndexDefs* de este modo:

Indice	Campos	Atributos
<i>Nombre</i>	<i>Nombre</i>	<i>ixCaseInsensitive</i>
<i>Apellidos</i>	<i>Apellidos</i>	<i>ixCaseInsensitive</i>

El próximo paso es crear el conjunto de datos en memoria, para lo cual pulsamos el botón derecho del ratón sobre *People*, y ejecutamos el comando *Create DataSet*. Este comando deja la tabla abierta, asignando *True* a *Active*. Aproveche y cree ahora los campos del conjunto de datos mediante el Editor de Campos. Asigne también a la propiedad *IndexFieldNames* el campo *Nombre*, para que los datos se presenten inicialmente ordenados por el nombre de la persona.

He dejado intencionalmente vacía la propiedad *FileName* del conjunto de datos. Mi propósito es redefinir el método *Loaded* del formulario, para asignar en tiempo de carga el nombre del fichero de trabajo:

```
void __fastcall TwndMain::Loaded()
{
    TForm::Loaded();
    People->FileName = ChangeFileExt(Application->ExeName, ".cds");
}
```

También es conveniente interceptar el evento *OnClose* del formulario, para mezclar los cambios y los datos originales antes de que el conjunto de datos cliente los guarde en disco:

```
void __fastcall TwndMain::FormClose(TObject *Sender,
    TCloseAction &Action)
{
    People->MergeChangeLog();
}
```

Los datos anidados se guardan de forma eficiente en la propiedad *Data*, es decir, sin repetir la cabecera. Pero cuando se encuentran en la propiedad *Delta*, es inevitable que se repita la cabecera. Por lo tanto, es conveniente mezclar periódicamente los cambios, con el fin de disminuir las necesidades de memoria.

No intente llamar a *MergeChangeLog* en el evento *BeforeClose* del conjunto de datos, pues al llegar a este punto ya se han guardado los datos en disco, al menos en la versión actual del componente.

Si en este momento mostramos todos los campos de *People* en una rejilla de datos, podemos editar el campo *Telefonos* pulsando el botón que aparece en la celda de la rejilla. Al igual que sucede con las tablas anidadas de Oracle 8, esta acción provoca la aparición de una ventana emergente con otra rejilla. Esta última contiene dos columnas: una para los números de teléfono y otra para los comentarios asociados.

Sin embargo, es más apropiado traer un segundo conjunto de datos cliente al formulario. Esta vez lo llamaremos *Phones*, y para configurarlo bastará asignar a su propiedad *DataSetField* el campo *PeopleTelefonos*. Cree campos persistentes para este con-

junto de datos, y asigne a *IndexFieldNames* el campo *Telefono*, para que estos aparezcan de forma ordenada.

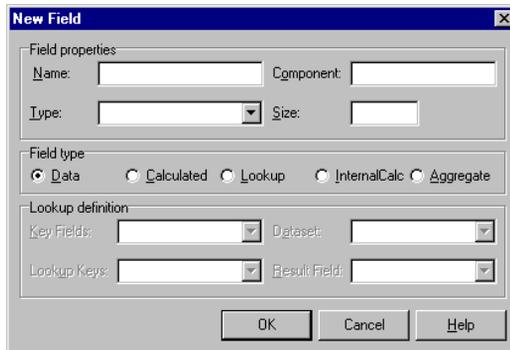
Es muy probable también que desee eliminar todos los teléfonos antes de eliminar a una persona²⁰, para lo que basta este método:

```
void __fastcall TwndMain::PeopleBeforeDelete(TDataSet *DataSet)
{
    Phones->First();
    while (! Phones->Eof)
        Phones->Delete();
}
```

Y ya puede definir formatos de visualización, comandos de edición y reglas de integridad al modo tradicional para completar este ejemplo.

Campos calculados internos

Por supuesto, podemos definir campos calculados y campos de búsqueda al utilizar conjuntos de datos clientes. Hay, sin embargo, un nuevo tipo de campo calculado que es típico de esta clase de componentes: los *campos calculados internos*. Para definir este tipo de campo, se utiliza el comando *New field*, del Editor de Campos:



Como podemos apreciar, el grupo de botones de radio que determina el tipo de campo tiene dos nuevas opciones: *InternalCalc* y *Aggregate*. La primera es la que nos interesa por el momento, pues la segunda se estudiará en la siguiente sección. El campo calculado interno se define marcando la opción *InternalCalc*, y especificando el nombre y tipo de datos del campo. El valor del campo se calcula durante la respuesta al evento *OnCalcFields* del conjunto de datos.

²⁰ Le apuesto una cena a que al leer la frase anterior pensó en alguien de su entorno. ¿No? Bueno, yo sí.

¿En qué se diferencian los campos calculados internos de los ya conocidos? Simplemente, en que el valor de los campos calculados internos se almacena también junto al conjunto de datos. La principal consecuencia de esta política de almacenamiento es que pueden definirse índices basados en estos campos. Así que tenemos algo parecido a índices por expresiones para los conjuntos de datos clientes.

Una información importante: aunque ambos tipos de campos calculados reciben sus valores en respuesta al evento *OnCalcFields*, en realidad este evento se dispara dos veces, una para los campos calculados internos y otra para los normales. Para diferenciar entre ambas activaciones, debemos preguntar por el valor de la propiedad *State* del conjunto de datos. En un caso, valdrá *dsCalcFields*, mientras que en el otro valdrá *dsInternalCalc*. El código de respuesta al evento puede evitar entonces realizar cálculos innecesarios.

Índices, grupos y valores agregados

Existen tres formas de crear índices en un conjunto de datos cliente:

- Utilizando la propiedad *IndexDefs*, antes de crear el conjunto de datos.
- Dinámicamente, cuando ya está creado el conjunto de datos, utilizando el método *AddIndex*. Estos índices sobreviven mientras el conjunto de datos está activo, pero no se almacenan en disco.
- Dinámicamente, especificando un criterio de ordenación mediante la propiedad *IndexFieldNames*. Estos índices desaparecen al cambiar el criterio de ordenación.

Cuando utilizamos alguna de las dos primeras técnicas, podemos utilizar opciones que solamente están disponibles para este tipo de índices:

- Para claves compuestas, algunos campos pueden ordenarse ascendentemente y otros en forma descendente.
- Para claves compuestas, algunos campos pueden distinguir entre mayúsculas y minúsculas, y otros no.
- Puede indicarse un nivel de agrupamiento.

Para entender cómo es posible indicar tales opciones, examinemos los parámetros del método *AddIndex*:

```
void __fastcall TClientDataSet::AddIndex(
    const AnsiString Name, const AnsiString Fields,
    TIndexOptions Options,
    const AnsiString DescFields = "";
    const AnsiString CaseInsFields = "";
    const int GroupingLevel = 0);
```

Los nombres de los campos que formarán parte del índice se pasan en el parámetro *Fields*. Ahora bien, podemos pasar en *DescFields* y en *CaseInsFields* un subconjunto de estos campos, indicando cuáles deben ordenarse en forma descendente y cuáles deben ignorar mayúsculas y minúsculas. Por ejemplo, el siguiente índice ordena ascendentemente por el nombre, teniendo en cuenta las mayúsculas y minúsculas, de modo ascendente por el apellido, pero sin distinguir los tipos de letra, y de modo descendente por el salario:

```
ClientDataSet1->AddIndex("Indice1", "Nombre;Apellidos;Salario",
    "Salario", "Apellidos");
```

El tipo *TIndexDef* utiliza las propiedades *DescFields*, *CaseInsFields* y *GroupingLevel* para obtener el mismo efecto que los parámetros correspondientes de *AddIndex*.

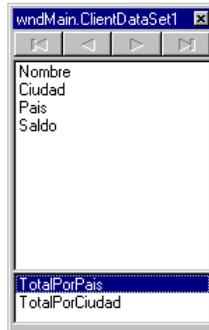
¿Y qué hay acerca del nivel de agrupamiento? Este valor indica al índice que dispare ciertos eventos internos cuando alcance el final de un grupo de valores repetidos. Tomemos como ejemplo el conjunto de datos que mostramos en la siguiente imagen:

País	Ciudad	Nombre	Saldo	Total por país	Total por ciudad
British West Indies	Grand Cayman	Cayman Divers World Unlimited	\$59.660,05	76511,8	76511,8
		Fisherman's Eye	\$12.022,00		
		Safari Under the Sea	\$4.829,75		
Canada	Kitchener	Marmot Divers Club	\$12.223,25	97358,9	12223,25
	Vancouver	Davy Jones' Locker	\$44.073,65		
	Winnipeg	Orn-Target SCUBA	\$41.062,00		
Columbia	Bogota	Fantastique Aquatica	\$90.143,40	90143,4	90143,4
Cyprus	Kato Paphos	Sight Diver	\$261.575,80	261575,8	261575,8
Fiji	Suva	Divers-for-Hire	\$21.534,00	37076	21534
	Taveuni	Princess Island SCUBA	\$15.542,00		15542
Greece	Ayios Matthaïos	Divers of Corfu, Inc.	\$98.278,60	98278,6	98278,6
Republic So. Africa	Johannesburg	Central Underwater Supplies	\$6.675,95	6675,95	6675,95
US	Catalina Island	Catamaran Dive Club	\$52.703,55	1494901,8	52703,55
	Eugene	Frank's Divers Supply	\$20.602,00		20602

En realidad, solamente las cuatro primeras columnas pertenecen al conjunto de datos: el país, la ciudad, el nombre de un cliente y el estado de su cuenta con nosotros. El índice que está activo, llamado *Jerarquia*, ordena por los tres primeros campos, y su *GroupingLevel* es igual a 2. De esta manera, podremos mantener estadísticas o agregados por los dos primeros campos del índice: el país y la ciudad.

Para activar el agrupamiento debemos definir *agregados* (*aggregate*s) sobre el conjunto de datos. Hay dos formas de hacerlo: definir los agregados en la propiedad *Aggregates*, o definir campos agregados en el Editor de Campos. En nuestro ejemplo seguiremos la segunda vía. En el Editor de Campos ejecutamos el comando *New field*. Como recordará de la sección anterior, ahora tenemos la opción *Aggregate* en el tipo de campo. Como nombre, utilizaremos *TotalPorPaís*. Repetiremos la operación para crear

un segundo campo agregado: *TotalPorCiudad*. En el Editor de Campos, estos dos campos se muestran en un panel situado en la parte inferior del mismo:



Debemos cambiar un par de propiedades para los objetos recién creados:

	TotalPorPais	TotalPorCiudad
Expression	<i>Sum(Saldo)</i>	<i>Sum(Saldo)</i>
IndexName	<i>Jerarquia</i>	<i>Jerarquia</i>
GroupingLevel	<i>1</i>	<i>2</i>

Vamos ahora a la rejilla de datos, y creamos un par de columnas adicionales, a la que asignamos explícitamente el nombre de los nuevos campos en la propiedad *FieldName* (C++ Builder no muestra los campos agregados en la lista desplegable de la propiedad). Para completar el ejemplo, debemos asignar *False* en la propiedad *DefaultDrawing* de la rejilla, y dar respuesta su evento *OnDrawColumnCell*:

```
void __fastcall TwndMain::DBGrid1DrawColumnCell(TObject *Sender,
const TRect Rect, int DataCol, TColumn Column,
TGridDrawState State);
{
    if (Column->Field == ClientDataSet1Pais ||
        Column->Field == ClientDataSet1TotalPorPais)
        if (! ClientDataSet1->GetGroupState(1).Contains(gbFirst))
            return;
    if (Column->Field == ClientDataSet1Ciudad ||
        Column->Field == ClientDataSet1TotalPorCiudad)
        if (! ClientDataSet1->GetGroupState(2).Contains(gbFirst))
            return;
    DBGrid1->DefaultDrawColumnCell(Rect, DataCol, Column, State);
}
```

Aquí hemos aprovechado la función *GetGroupState* del conjunto de datos. Esta función devuelve un conjunto con los valores *gbFirst*, *gbMiddle* y *gbLast*, para indicar si estamos al inicio, al final o en mitad de un grupo.

El Modelo de Objetos Componentes

EN ESTE CAPÍTULO INICIAMOS EL ESTUDIO DEL denominado *Modelo de Objetos Componentes* o COM, según sus siglas en inglés. El Modelo de Objetos Componentes es una de las armas más potentes con las que cuenta el programador de Windows para desarrollar aplicaciones distribuidas. COM es, además, uno de los pilares en los que se basa el desarrollo con Midas y con Microsoft Transaction Server. Desafortunadamente, potencia equivale en este caso a complejidad. Aunque los conceptos de la programación COM son muy sencillos, la forma en que se plasman en lenguajes “históricos” como C y C++ deja mucho que desear.

Es por eso que he dividido el material sobre COM en varios capítulos. Aquí desarrollaremos los conceptos básicos y le enseñaremos a trabajar con objetos creados por otros.

Un modelo binario de objetos

La cercanía y la familiaridad son el peor enemigo de la perspectiva. Muchas veces no apreciamos correctamente aquello que utilizamos a diario, y esta vez me refiero a las Bibliotecas de Enlace Dinámico, o DLLs, no a la belleza de su cónyuge. El programador de Windows se ha acostumbrado a este recurso que le permite ejecutar desde su programa, desarrollado en el lenguaje *alfa*, funciones y procedimientos desarrollados en un lenguaje *beta*. Tal es la costumbre que pocas veces meditamos en qué es lo que hace posible esta comunicación tan sencilla y transparente. ¿La respuesta?, la existencia de unas reglas de juego que deben ser respetadas por el que crea la DLL y por aquellos que la usan. Unas reglas de juego que son, nada más y nada menos, que un modelo binario (o físico) de intercambio de información. ¿Quiere ver algunas de las reglas para las DLLs? Por ejemplo:

1. Los parámetros de las funciones se copian en la pila de derecha a izquierda.
2. El espacio de memoria reservado para los parámetros es liberado por el que implementa la función, no por quien la llama.

3. Si usted implementa una función que debe recibir información alfanumérica, es aconsejable que utilice como parámetro un puntero a una lista de caracteres que termine con el carácter nulo. Su lenguaje de programación posiblemente tenga una mejor representación para este tipo de datos, pero usted tendrá que renunciar a ella si quiere que la función pueda ser llamada desde un lenguaje arbitrario.

Poder compartir y reutilizar funciones y procedimientos es algo muy positivo. Pero cualquier programador medianamente informado conoce las innumerables ventajas que ofrece la Programación Orientada a Objetos sobre la programación tradicional basada en funciones. ¿Qué tal si pudiéramos compartir objetos con la misma facilidad con la que usamos una DLL? Necesitaríamos entonces un estándar binario para el intercambio de objetos. Y ese, precisamente, es el propósito de COM: el *Modelo de Objetos Componentes* o, en inglés, *Component Object Model*.

De modo que COM nos permitirá reutilizar objetos sin preocuparnos del lenguaje en que han sido creados, compilados y enlazados. Naturalmente, las exigencias de un modelo binario de objetos van más allá de la simple independencia del lenguaje. Hace falta también la independencia de localización, de modo que un proceso pueda utilizar transparentemente objetos ubicados en ordenadores remotos. Veremos también cómo COM resuelve este problema.

¡Yo quiero ver código!

El peor error que puede cometer un escritor es no confiar en la inteligencia de sus lectores²¹. Cuando escribí este capítulo por primera vez, comencé explicando todos los conceptos teóricos, dejando para el final los ejemplos. Llegó el momento de la revisión: en un par de ocasiones intenté leer lo que había escrito, y en ambas ocasiones me quedé dormido a mitad del intento. Tanta teoría sin ejemplos puede agotar hasta la infinita paciencia del presidente de la ONU.

Esta sección es un añadido posterior. Aquí le presento de golpe un ejemplo de función que crea un objeto COM, realiza un par de llamadas a sus métodos y lo libera finalmente. ¿El objetivo? Crear un icono de acceso directo a un programa.

```
// Necesita incluir la cabecera shlobj.h

void CreateLink(const AnsiString& APath, const WideString& AFile)
{
    IUnknown *intf;
    IShellLink *slink;
    IPersistFile *pfile;

    OleCheck(CoCreateInstance(CLSID_ShellLink, NULL,
```

²¹ El siguiente consiste en confiar demasiado.

```

        CLSCTX_ALL, IID_IUnknown, (void**) &intf));
    try
    {
        OleCheck(intf->QueryInterface(IID_IShellLink,
            (void**) &slink));
        try
        {
            slink->SetPath(APath.c_str());
            OleCheck(intf->QueryInterface(IID_IPersistFile,
                (void**) &pfile));
            try
            {
                pfile->Save(AFile, False);
            }
            finally
            {
                pfile->Release();
            }
        }
        finally
        {
            slink->Release();
        }
    }
    finally
    {
        intf->Release();
    }
}

```

¿Se entera de algo? En caso afirmativo, cierre el libro y saque a su cocodrilo a pasear por el parque, que es mejor para la salud que leer libros de Informática. En caso contrario, tenga fe en que al terminar el capítulo no sólo sabrá de qué va el asunto, sino que podrá escribir una versión más reducida de *CreateLink* y enviármela por correo.

Clases, objetos e interfaces

En este juego hay dos protagonistas principales: un cliente y un servidor. Como es de esperar, el servidor es quien implementa los objetos a compartir, y el cliente es el que los utiliza. Estos roles no son fijos, pues es posible que un cliente de determinado servidor sea, a su vez, el servidor de otro cliente. Incluso se puede dar el caso en que dos procesos actúen mutuamente como cliente y servidor: ambos procesos definen objetos que son utilizados por la parte contraria.

Ahora, para comenzar a familiarizarnos con la jerga de COM, veamos las fichas que mueven los personajes anteriores: las clases, los objetos y las interfaces.

Interfaces: El concepto básico del modelo es el concepto de *interfaz*. Una interfaz representa un conjunto de funciones que se ponen a disposición del público. Por ejemplo, una máquina de bebidas puede modelarse mediante funciones para:

- Introducir monedas.
- Seleccionar producto.
- Extraer producto.
- Recuperar monedas.

No se trata de funciones sin relación entre sí, pues el objeto que ofrece estos servicios debe recordar, al ejecutar el segundo paso, cuántas monedas hemos introducido en el primer paso. Esta información, como es clásico en la Programación Orientada a Objetos, debe almacenarse en variables internas del objeto que implementa los servicios.

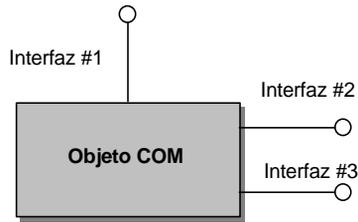
Clases: Es importante comprender que una interfaz debe ser una especificación *abstracta*, que debe haber libertad en su implementación y que al usuario de los servicios le debe dar lo mismo hacer uso de una u otra implementación alternativa. Una *clase* es una entidad conceptual que puede ofrecer a sus clientes potenciales una o más interfaces.

Objetos: Para usted, que ya está familiarizado con la Programación Orientada a Objetos, la distinción entre *clase* y *objeto* le será más que conocida. Por lo tanto, no insistiré en este punto.

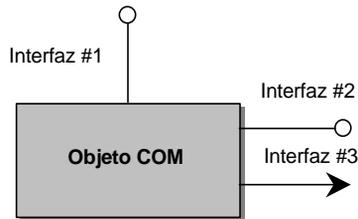
Naturalmente, para crear un objeto hay que especificar una clase (no una interfaz). Como consecuencia, cada objeto COM puede ser percibido y tratado por sus clientes de diversas formas, de acuerdo al número de interfaces que implementa la clase a la cual pertenece. Por ejemplo, una clase que implemente la interfaz *IMaquinaBebida* posiblemente implemente también la interfaz *IAparatoElectrico*, con métodos para enchufar, desenchufar y calcular consumo. A un cliente sediento solamente le importará la primera interfaz. En cambio, al dueño del local donde está la máquina le interesa principalmente la interfaz de aparato eléctrico, lo cual no quita que de vez en cuando se beba una cerveza de la máquina, a la salud del Modelo de Objetos Componentes.

En el ejemplo de la sección anterior, se crea un *objeto* de la *clase* identificada mediante la constante *CLSID_ShellLink*. Las variables locales *intf*, *slink* y *pfile* son punteros a interfaces, que se utilizan para ejecutar los métodos exportados por el objeto creado.

En la mayoría de los artículos y libros sobre COM, se utiliza un diagrama parecido al siguiente para representar un objeto de este modelo:



La simbología es explícita: un objeto COM es una caja negra, a la cual tenemos acceso únicamente a través de un conjunto de interfaces, representadas mediante las líneas terminadas con un círculo. El círculo hueco significa que los clientes pueden “enchufarse” a estas interfaces para llamar los métodos ofrecidos por ellas. La presencia de los “huecos” no es trivial ni redundante; más adelante veremos cómo los objetos COM pueden implementar interfaces “de salida” (*outgoing interfaces*), que nos servirán para recibir eventos disparados por servidores COM. En tales casos, la interfaz de salida se representa mediante una flecha convencional:



Por el momento, mantendré en secreto el motivo por el que una de las “antenas” de nuestro insecto apunta a las alturas.

El lenguaje de descripción de interfaces

Es hora de ser más precisos. Hasta el momento hemos mencionado interfaces como servicios, y clases como entidades que implementan uno o más de estos servicios, pero no hemos presentado ni una sola línea de código. ¿Qué tal si le presento como ejemplo la definición de una máquina de bebidas? Usted espera que yo le muestre entonces una declaración en C++ que corresponda a la estructura de una interfaz. Pero, ¿no habíamos quedado en que los objetos COM pueden utilizarse desde cualquier lenguaje de programación?

Queda claro que debe existir un lenguaje *neutro* para describir las posibilidades de las interfaces y clases. Debe permitir expresar todas las características de la descripción de un objeto COM que sean comunes a cualquier lenguaje. Y el lenguaje que utiliza COM se llama *IDL*, por las iniciales de *Interface Description Language*. El IDL de COM es en realidad una variación del lenguaje del mismo nombre que se utiliza en RPC para describir procedimientos remotos.

El objetivo principal de IDL no es generar programas, sino descripciones. Se supone que, utilizando las herramientas adecuadas, una especificación en IDL puede traducirse mecánicamente a declaraciones del lenguaje en el que deseemos desarrollar realmente. En particular, los programadores de C++ Builder no tienen por qué escribir nada en IDL, pues pueden utilizar un editor visual para definir clases e interfaces: el Editor de Bibliotecas de Tipos, que veremos más adelante.

Veamos primero el aspecto de IDL, para ocuparnos después de cómo podemos sacar partido del lenguaje. Comenzaremos la definición de clases:

```
coclass MaquinaBebidas
{
    [default] interface IMaquinaBebidas;
    interface IAparatoElectrico;
};
```

La palabra **coclass** se utiliza para declarar clases de objetos componentes; se trata únicamente de un problema de terminología. La descripción pública de una clase de componentes sencillamente consiste en enumerar las interfaces que implementa. En nuestro caso, se implementan dos interfaces: *IMaquinaBebidas* y *IAparatoElectrico*, las cuales deben haberse definido antes. Como convenio, los tipos de interfaces comienzan con la letra *I*.

El atributo **default** asociado a la primera interfaz indica que ésta es la interfaz que representa “de forma más natural” la funcionalidad de la clase asociada. Cuando se crea un objeto de dicha clase, casi siempre se intenta suministrar al cliente inicialmente un puntero a dicha interfaz (no siempre es posible). Por supuesto, se trata de recomendaciones de carácter semántico. Este es el motivo por el que, en un diagrama anterior, un objeto COM era dibujado con una de sus “antenas” erectas.

La siguiente declaración simplificada muestra cómo definir una interfaz:

```
interface IMaquinaBebidas
{
    HRESULT IntroducirMonedas([in] long Cantidad);
    HRESULT SeleccionarProducto([in] BSTR NombreProducto);
    HRESULT ExtraerProducto(void);
    HRESULT RecuperarMonedas([out] long* Cantidad);
};
```

A simple vista se aprecia la semejanza de las declaraciones de interfaces con las declaraciones de clases de C/C++. Los métodos definidos en las interfaces, sin embargo, no pueden indicar una implementación. Además, IDL añade información semántica que no se puede expresar en C++. Por ejemplo, el parámetro de *RecuperarMonedas*

está marcado con el atributo **out**, mientras que nada impide que una rutina en C++ utilice un parámetro de puntero para leer y escribir. En cuanto al tipo *HRESULT* que retornan los métodos de la interfaz, me reservo su explicación para otro momento más adecuado.

En general, toda la información semántica de IDL que no puede expresarse de forma directa en la mayoría de los lenguajes de programación, se coloca dentro de corchetes, a modo de atributos.

Identificadores globales únicos

Para poder compartir definiciones de clases e interfaces entre aplicaciones escritas en diferentes lenguajes se necesita algún modo de identificar a estas entidades. Si necesitamos pedir a un servidor que cree un objeto de la clase *MaquinaBebidas*, podemos vernos tentados a utilizar el nombre de la clase como identificación de la misma. El mecanismo básico de COM, sin embargo, no se basa en la identificación mediante cadenas de caracteres. El principal motivo es que podrían aparecer colisiones de nombres con suma facilidad. Por el contrario, COM identifica a las clases e interfaces mediante valor numéricos únicos de 128 bits.

A este tipo de números se les conoce como *GUIDs* (*Global Unique Identifiers*, o identificadores únicos globales). Cuando definimos una clase o una interfaz con IDL debemos asociarle uno de estos números, para lo cual se habilita una sección de atributos previa a la definición de la entidad:

```
[
    uuid(CADC5FC4-D4A8-11D2-B67D-0000E8D7F7B2)
]
coclass MaquinaBebidas
{
    [default] interface IMaquinaBebidas;
    interface IAparatoElectrico;
};
```

Los guiones y llaves son pura parafernalia mnemotécnica, por si algún chalado es capaz de recordarlos. Si un GUID se utiliza para identificar una interfaz, se le aplica el nombre especial de IID; si identifica a una clase, estamos frente a un CLSID.

Si usted o yo definimos una clase o interfaz, debemos asignarle un número único de éstos. Podemos pedirle uno a Microsoft, de forma similar en que las direcciones de Internet se piden a la organización INTERNIC; realmente, Microsoft lo tiene más fácil, al tratarse de 128 bits en vez de 32. Pero los GUID también pueden generarse localmente, por medio de una función definida en el API de Windows, *CoCreateGuid*. Esta función genera números con *unicidad estadística*; vamos, que la probabilidad de un

conflicto es menor que la de que acertemos la lotería. El algoritmo empleado introduce la fecha y la hora dentro de un sombrero de copa, le añade el identificador de red del usuario, o información del BIOS, si no estamos en red, realiza un par de pases con la varita mágica y saca un conejo con un GUID en la boca.

La siguiente función devuelve como resultado el número de la tarjeta de red del ordenador donde se ejecuta. Este número es un entero de 48 bits, por lo cual me he limitado a devolver la representación en hexadecimal del mismo:

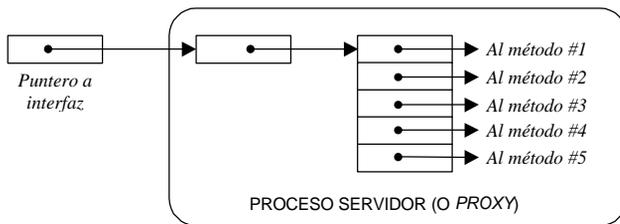
```
 AnsiString __fastcall NetworkId()
 {
     _GUID guid;

     CoCreateGuid(&guid);
     AnsiString rslt = GUIDToString(guid);
     return rslt.SubString(rslt.Length() - 12, 12);
 }
```

La mayoría de las veces, C++ Builder genera por nosotros los identificadores de interfaces y otros tipos de GUID utilizados por COM. Pero si en algún momento necesita uno de estos números, pulse la combinación CTRL+MAY+G en el Editor de Código.

Interfaces

Para el Modelo de Objetos Componentes es fundamental el concepto de *interfaz*. Sabemos que una de las técnicas fundamentales de la programación consiste en denegar al programador el acceso a la implementación de un recurso; de esta forma lo protegemos de los cambios que ocurrirán muy probablemente en la misma. Consecuentemente con esta filosofía, en COM usted nunca trabajará con un objeto COM, ni con el puntero al objeto. Por el contrario, con lo que podrá contar es con *punteros a interfaces*.



Al ser COM precisamente un modelo binario, es muy importante saber cómo deben implementarse físicamente las interfaces. Resulta que una interfaz se representa como

un puntero a una tabla de punteros a métodos, similar a la Tabla de Métodos Virtuales de los objetos de C++.

¿Por qué un puntero a una tabla de punteros a funciones? Porque esta es la implementación más frecuente de un objeto en lenguajes que permiten sólo herencia simple. Supongamos que los objetos de una clase COM implementan una sola interfaz. En tal caso, el implementador de la clase puede utilizar el formato de objetos con herencia simple, y el puntero al objeto es válido igualmente como puntero a la interfaz. Sin embargo, no se deje engañar por esta aparente simplicidad. Es habitual que un objeto COM resida en un espacio de memoria diferente al del cliente que lo utiliza. Por ese motivo, en el diagrama anterior he utilizado la misteriosa palabra *proxy*, que explicaremos más adelante.

Cada lenguaje de programación representará los punteros a interfaces mediante sus propios tipos nativos. En la medida en que el lenguaje esté más o menos preparado para este propósito, la traducción será más o menos legible. La forma más común de obtener una representación nativa es utilizar alguna herramienta que traduzca automáticamente desde IDL hacia el lenguaje que queremos utilizar. Si estamos trabajando con C/C++, Microsoft nos ofrece un compilador denominado MIDL, que se encarga de esta tarea. Pero si desarrollamos con C++ Builder, la traducción se puede realizar desde el Editor de Bibliotecas de Tipos, que explicaremos más adelante.

¿Recuerda nuestra declaración de la interfaz de una máquina de bebidas? Su representación en C++ sería:

```
interface IMaquinaBebidas : public IUnknown
{
public:
    virtual HRESULT STDMETHODCALLTYPE IntroducirMonedas
        (long Cantidad) = 0;
    virtual HRESULT STDMETHODCALLTYPE SeleccionarProducto
        (BSTR NombreProducto) = 0;
    virtual HRESULT STDMETHODCALLTYPE ExtraerProducto
        (void) = 0;
    virtual HRESULT STDMETHODCALLTYPE RecuperarMonedas
        (long* Cantidad) = 0;
};
```

La declaración comienza con la macro *interface*, que se define del siguiente modo en el fichero *objbase.h*:

```
#define interface struct
```

Como la estructura definida tiene métodos virtuales, el compilador de C++ crea automáticamente una Tabla de Métodos Virtuales (VMT), y cada instancia de esta estructura comienza precisamente con un puntero a dicha VMT. Un programa es-

crito en C++ que trabajara con esta interfaz tendría más o menos el siguiente aspecto:

```

{
    IMaquinaBebidas *maq;      // Observe que se trata de un puntero

    // -----
    // Crear un objeto y obtener puntero a la interfaz
    // -----
    maq->IntroducirMonedas(100);
    maq->SeleccionarProducto(L"Coca-Cola");
    maq->ExtraerProducto();
    // -----
    // Liberar el puntero a la interfaz
    // -----
}

```

Las representaciones de interfaces en lenguajes más modernos, como Delphi y Java, tienen un aspecto más “natural”, porque no hacen tanto uso de macros. Estos dos lenguajes ofrecen un tipo nativo **interface** que hace más fácil el trabajo con punteros a interfaces.

La interfaz *IUnknown*

En la sección anterior, al presentar la declaración de la interfaz *IMaquinaBebidas* en C++ se nos escapó un pequeño adelanto:

```
interface IMaquinaBebidas : public IUnknown
```

Al parecer, la estructura que representa a nuestra interfaz hereda de una clase o estructura que no hemos aún explicado. Bien, *IUnknown* es una interfaz predefinida por COM a partir de la cual se derivan obligatoriamente todas las demás interfaces. Veamos en primer lugar la definición de la interfaz *IUnknown* para explicar después cómo es posible que una interfaz derive de otra. Esta es la definición de *IUnknown* en el lenguaje IDL:

```

[ local, object,
  uuid(00000000-0000-0000-C000-000000000046),
  pointer_default(unique)
]
interface IUnknown
{
    HRESULT QueryInterface(
        [in] REFIID riid,
        [out, iid_is(riid)] void **ppvObject);
    ULONG AddRef();
    ULONG Release();
}

```

Olvidémonos por un momento de los atributos que acompañan a la definición, para analizar qué servicios nos ofrece esta interfaz básica. Son tres los métodos exportados por *IUnknown*, y sus objetivos son:

- *QueryInterface*: Aporta introspección a los objetos. Dada una interfaz cualquiera, nos dice si el objeto asociado soporta otra interfaz determinada; en caso afirmativo, nos devuelve el puntero a esa interfaz. Si tenemos un puntero a una interfaz de tipo *IMaquinaBebidas* en nuestras manos, por ejemplo, podemos aprovechar *QueryInterface* para averiguar si el objeto asociado soporta también la interfaz *IAparatoElectrico*.
- *AddRef*: Debe incrementar un contador de referencias asociado al objeto. En conjunción con el siguiente método, se utiliza para controlar el tiempo de vida de los objetos COM.
- *Release*: Debe decrementar el contador de referencias, y destruir el objeto si el contador se vuelve cero.

Existe una relación de herencia simple entre interfaces. Tomemos como ejemplo la siguiente definición:

```
[   object, uuid(0000011b-0000-0000-c000-000000000046),
    pointer_default(unique)
]
interface IOleContainer : IParseDisplayName
{
    HRESULT EnumObjects([in] DWORD grfFlags,
        [out] IEnumUnknown **ppenum);
    HRESULT LockContainer([in] BOOL fLock);
}
```

Según la declaración, la interfaz *IOleContainer* debe ofrecer todos los métodos de la interfaz *IParseDisplayName*, sean los que sean, más los métodos *EnumObjects* y *LockContainer*. Cuando una declaración de interfaz no menciona ninguna interfaz madre, se asume que ésta es *IUnknown*. Por lo tanto, todas las interfaces ofrecen obligatoriamente los tres métodos *AddRef*, *Release* y *QueryInterface*.

Tiempo de vida

Los lenguajes más modernos, como Java y Delphi, tienen tipos nativos que sirven para representar de forma transparente los punteros a interfaces. Los lenguajes mencionados se ocupan automáticamente de la liberación de los objetos COM, para lo cual el compilador genera llamadas a los métodos *AddRef* y *Release* de las interfaces con que trabaja. Pero en los lenguajes tradicionales como C y C++, tenemos que ocuparnos explícitamente de llamar a los métodos *AddRef* y *Release* para que el objeto COM asociado sepa cuándo destruirse. Por lo tanto, debemos aprender bien las reglas relacionadas con el tiempo de vida.

1. Las funciones que devuelven un puntero a interfaz ya han incrementado el contador de referencia del objeto subyacente. Usted no debe llamar a *AddRef*.
2. Cuando se asigna una variable que apunta a una interfaz en otra, se debe llamar a *AddRef*.
3. Cuando se termina el tiempo de vida de una variable de puntero a interfaz, o se va a sobrescribir su valor, debemos llamar a *Release*.

En nuestro ejemplo de creación de accesos directos la mayor parte de las líneas se dedican a garantizar el mantenimiento de las interfaces. Es fácil identificar el siguiente patrón de programación:

```
ITipoInterfaz* interfaz = PedirInterfaz();
try
{
    // Utilizar la interfaz
}
finally
{
    interfaz->Release();
}
```

En realidad, la mayor parte de las líneas de la función *CreateLink* se han dedicado a asegurar la liberación del objeto creado. Dentro de muy poco presentaremos técnicas para facilitar esta tarea.

Introspección

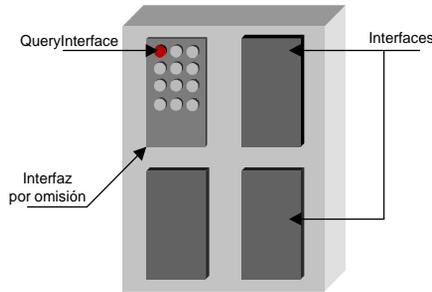
El siguiente símil le será de utilidad para comprender la relación entre clases e interfaces en COM: un objeto COM es una caja negra con varios paneles de botones. Cada panel representa una interfaz, y los botones son los métodos declarados por esa interfaz. El problema consiste en que cada panel tiene una tapa independiente, y en que inicialmente solamente está descubierta la tapa de un solo panel (*IUnknown*). Para abrir la tapa de cualquier otro panel, debe pulsar un botón especial, que está presente en todos los paneles (*QueryInterface*).

En el ejemplo del inicio del capítulo llamamos a *QueryInterface* en dos ocasiones diferentes. La primera es la siguiente:

```
OleCheck(intf->QueryInterface(IID_IShellLink, (void**) &slink));
```

En primer lugar, veamos el control de errores. Casi todas las funciones de COM utilizan códigos para señalar incumplimientos de contrato. Los códigos se representan mediante el tipo HRESULT, que contiene enteros de 32 bits. El valor especial *S_OK* indica que no hubo problemas con la función. *OleCheck* es un procedimiento

específico de C++ Builder, que recibe un valor de tipo *HRESULT* y lanza una excepción, como mandan las normas, si hay problemas.



En cuanto a *QueryInterface* en sí, recibe como primer parámetro un identificador único de interfaz, mientras que el segundo parámetro es un puntero a un puntero a interfaz. En C++ moderno quizás hubiéramos utilizado una referencia a un puntero a interfaz, aunque hubiéramos perdido la posibilidad de pasar *0* en este parámetro. Observe que hay que convertir “a lo bestia” el puntero obtenido en un puntero a puntero a **void**. Cosas de este lenguaje.

Existen unas reglas del juego que cualquier implementación de *QueryInterface* debe satisfacer. Este método define una relación matemática de equivalencia entre las interfaces soportadas por un mismo objeto. Supongamos que un objeto implementa las interfaces *IID_I1*, *IID_I2* e *IID_I3*. Entonces se cumple que la relación entre interfaces determinada por *QueryInterface* es:

- **Simétrica:** Si la interfaz *I1* puede obtenerse a partir de un puntero a la interfaz *I2*, se podrá también obtener la interfaz *I2* a partir de un puntero a *I1*:

```
if (I1->QueryInterface(IID_I2, (void**) &I2) !=
    I2->QueryInterface(IID_I1, (void**) &I1))
    ShowMessage(";Imposible!");
```

- **Transitiva:** Si de la interfaz *I1* puede obtenerse un puntero a *I2*, y si desde *I2* puede obtenerse un puntero a *I3*, también se podrá obtener directamente desde *I1* un puntero a *I3*.

```
if (I1->QueryInterface(IID_I2, (void**) &I2) == S_OK &&
    I2->QueryInterface(IID_I3, (void**) &I3) == S_OK &&
    I1->QueryInterface(IID_I3, (void**) &I3) != S_OK)
    ShowMessage(";Imposible!");
```

- **Reflexiva:** A partir de cualquier puntero a interfaz, siempre puede volver a obtenerse ella misma.

```
if (I1->QueryInterface(IID_I1, (void**) &I1) != S_OK)
    ShowMessage("¡Imposible!");
```

Cómo obtener un objeto COM

Para que una aplicación pueda crear un objeto definido en otro módulo, sea ejecutable o DLL, necesitamos funciones y procedimientos definidos por COM. La función más sencilla que permite crear un objeto COM es *CoCreateInstance*.

```
STDAPI CoCreateInstance(REFCLSID rclsid, LPUNKNOWN pUnkOuter,
    DWORD dwClsContext, REFIID riid, LPVOID *ppv);
```

Cada parámetro tiene el siguiente significado:

- *rclsid*: Aquí pasamos el identificador de la clase del objeto que deseamos crear.
- *pUnkOuter*: COM ofrece soporte especial para la *agregación* de objetos. En este modelo, un objeto “externo” pueda administrar el tiempo de vida de uno o más objetos “internos”, a la vez que permite al cliente trabajar directamente con punteros a interfaces de los subobjetos.
- *dwClsContext*: Indica qué tipo de servidor deseamos utilizar. Más adelante veremos que un servidor puede implementarse mediante un DLL o un ejecutable local o remoto.
- *riid*: Una vez creado el objeto, se busca determinada interfaz en su interior, para devolver el puntero a la misma. Este parámetro sirve para identificar dicha interfaz. En la mayoría de los casos se utiliza *IID_IUnknown*.
- *ppv*: Es un puntero a un puntero a una interfaz que sirve para depositar el puntero a la interfaz localizada por *CoCreateInstance*.

La siguiente instrucción es utilizada por el ejemplo del principio del capítulo para obtener el objeto COM. Se pide un objeto de la clase *CLSID_ShellLink*; la declaración de esta constante reside en el fichero *shlobj.h*. El objeto que se crea no formará parte de un agregado (el *NULL* en el segundo parámetro). Nos da lo mismo el tipo de servidor (*CLSCTX_ALL*). El puntero de interfaz inicial debe ser de tipo puntero a *IUnknown*; más eficiente hubiera sido pedir un puntero a la interfaz *IShellLink*, pero he seguido el camino más largo para ilustrar mejor el uso de COM.

```
OleCheck(CoCreateInstance(CLSID_ShellLink, NULL,
    CLSCTX_ALL, IID_IUnknown, (void**) &intf));
```

C++ Builder ofrece una función más conveniente para determinadas situaciones:

```
_di_IUnknown CreateComObject(const GUID& ClassID);
```

El tipo *_di_IUnknown* lo define C++ Builder como un “puntero inteligente” a la interfaz *IUnknown*; en la siguiente sección explicaré de qué se trata. *CreateComObject* crea un objeto de la clase *ClassID*, no utiliza agregación, requiere un servidor local, y la interfaz inicial que busca es la de tipo *IUnknown*.

Otra variante de creación de objetos COM la ofrece la siguiente función de la VCL:

```
_di_IUnknown CreateRemoteComObject(const WideString& MachineName,
    const GUID& ClassID);
```

Esta vez COM busca la información de la clase indicada en el registro de otra máquina, crea el objeto basado en la clase en el espacio de direcciones de ese ordenador, y devuelve nuevamente un “puntero” al objeto creado. En el siguiente capítulo veremos como DCOM hace posible esta magia.

Existen otras dos funciones importantes para la creación de objetos COM: *CoCreateInstanceEx*, que permite obtener varias interfaces a la vez durante la creación, y *CoGetClassObject*. Esta última es útil cuando se quieren crear varias instancias de la misma clase, pero tendremos que esperar al capítulo siguiente para explicar su funcionamiento.

Punteros inteligentes a interfaces

En el manual del programador de C++ Builder se nos intenta convencer de que el prefijo *_di_*, como el que se emplea en *_di_IUnknown*, quiere decir *dual interface*, o interfaz dual. Es mentira. En realidad quiere decir *Delphi interface*; en el capítulo sobre Automatización OLE explicaremos qué es en realidad una interfaz dual. La clave del asunto está en que C++ Builder define una clase *DelphiInterface* que, como seguro que ya usted sospecha, implementa punteros inteligentes a interfaces. Sin embargo, no voy a utilizar esta clase en mis ejemplos, pues solamente resuelve los problemas de referencias asociados a la copia de punteros, y no ofrece ningún tipo de encapsulamiento a las dos operaciones más frecuentes en COM: creación de objetos y llamadas a *QueryInterface*.

No obstante, es muy fácil crear nuestras propias clases alternativas de punteros inteligentes a interfaces. La siguiente clase, por ejemplo, está inspirada a medias en la *DelphiInterface* de Borland y en una clase homónima descrita en el excelente libro *Essential COM*, de Don Box²².

²² Addison-Wesley, 1998, ISBN 0-201-63446-5

```

template <class T> class TSmartIntf
{
private:
    // Datos privados de la clase
    T* t;
public:
    // Constructores
    __fastcall TSmartIntf();
    __fastcall TSmartIntf(T* intf);
    __fastcall TSmartIntf(const GUID& clsid);
    __fastcall TSmartIntf(IUnknown* intf, const GUID& iid);
    // El destructor de la clase
    __fastcall ~TSmartIntf();
    // El operador de asignación
    TSmartIntf<T> __fastcall operator=(T* intf);
    // Operadores de conversión y misceláneos
    operator T*() const;
    bool operator !() const;
    T** __fastcall operator &();
    T* __fastcall operator->() const;
};

```

He definido cuatro diferentes constructores:

```

template <class T> __fastcall TSmartIntf<T>::TSmartIntf() :
    intf(0) {}

template <class T> __fastcall TSmartIntf<T>::TSmartIntf(T* intf)
{
    if (intf != 0)
        intf->AddRef();
    t = intf;
}

template <class T> __fastcall TSmartIntf<T>::TSmartIntf(
    const GUID& clsid)
{
    OleCheck(CoCreateInstance(clsid, NULL, CLSCTX_ALL,
        IID_IUnknown, (void**) &t));
}

template <class T> __fastcall TSmartIntf<T>::TSmartIntf(
    IUnknown* intf, const GUID& iid)
{
    OleCheck(intf->QueryInterface(iid, (void**) &t));
}

```

El destructor de la clase verifica que el puntero a interfaz que contiene no sea nulo, para ejecutar el método *Release* sobre el mismo. Se asigna *0* al puntero para mayor seguridad:

```

template <class T> __fastcall TSmartIntf<T>::~TSmartIntf()
{
    if (t != 0) t->Release();
    t = 0;
}

```

También se sobrecarga el operador de asignación.

```
template <class T>
TSmartIntf<T> __fastcall TSmartIntf<T>::operator=(T* intf)
{
    if (intf != 0) intf->AddRef();
    if (t != 0) t->Release();
    t = intf;
    return *this;
}
```

¿Por qué no he definido un constructor que reciba como parámetro una referencia a la propia clase *TSmartIntf* (un constructor de copia), o una versión del operador de asignación que a la derecha pueda recibir una interfaz inteligente? Pues porque he incluido un operador que convierte a los objetos de tipo *TSmartIntf* en punteros a la interfaz subyacente, y porque ya existen versiones del constructor y de la asignación que acepta a este último tipo de datos:

```
template <class T> TSmartIntf<T>::operator T*() const
{
    return t;
}
```

El siguiente operador se utiliza para saber si el puntero a interfaz es o no es nulo:

```
template <class T> bool TSmartIntf<T>::operator !() const
{
    return (t != 0);
}
```

Por último, estos son los clásicos operadores que permiten delegar al puntero a interfaz interno las llamadas realizadas sobre la clase *TSmartIntf*:

```
template <class T> T** __fastcall TSmartIntf<T>::operator &()
{
    return &t;
}

template <class T> T* __fastcall TSmartIntf<T>::operator->() const
{
    return t;
}
```

¿Recuerda el ejemplo de creación de accesos directos? Utilizando la nueva clase, el código necesario adelgaza radicalmente:

```
void __fastcall CreateLink(const AnsiString& APath,
const WideString& AFile)
{
    TSmartIntf<IUnknown> i(CLSID_ShellLink);
    TSmartIntf<IShellLink> sl(i, IID_IShellLink);
    sl->SetPath(APath.c_str());
}
```

524 *La Cara Oculta de C++ Builder*

```
TSmartIntf<IPersistFile> pf(i, IID_IPersistFile);  
pf->Save(AFile, False);  
}
```

Servidores COM

EL GRAN DEFECTO DE C Y C++ CONSISTE en el uso y abuso de macros, que permiten crear un “lenguaje a la medida”, que solamente entienden el programador que se lo inventa y un puñado de celotes. Uno de los mejores ejemplos es la *ATL (ActiveX Template Library)*, la biblioteca de clases de Microsoft que supuestamente debe ayudarnos a programar para el modelo COM. Lamentablemente, C++ Builder basa la programación de servidores COM, en aras de la tan traída compatibilidad, en la ATL.

Sería demasiado complicado explicar todos los detalles de la programación con ATL a los recién llegados al mundo COM. Por lo tanto, en este capítulo se narra, desde lo más básico, la creación de servidores COM.

Interceptando operaciones en directorios

Cuando estudiamos algún tipo de sistema cliente/servidor (en el sentido más amplio, que va más allá de las bases de datos) y pensamos en algún ejemplo de programación, nuestras neuronas adoptan casi inmediatamente el papel de clientes. ¿Los servidores?, bueno, alguien los debe programar... Sin embargo, en la programación COM es bastante frecuente que tengamos que desempeñar el papel de servidores.

Por ejemplo, si queremos definir extensiones al Explorador, Windows nos pide que le suministremos clases COM que implementen determinadas interfaces predefinidas. Las extensiones más comunes son los menús de contexto. ¿Ha instalado alguna vez el programa WinZip en su ordenador? ¿Se ha fijado en que el menú que aparece cuando se pulsa el botón derecho del ratón sobre un fichero cualquiera tiene una nueva opción, para añadir a un archivo comprimido? Esto sucede porque WinZip instala y registra DLLs que actúan como servidores dentro del proceso, y que son utilizadas automáticamente por el Explorador de Windows. Otro ejemplo de extensión se conoce como *copy books* (¿ganchos de copia?), y son módulos que se activan cuando el usuario va a realizar algún tipo de operación sobre un directorio.

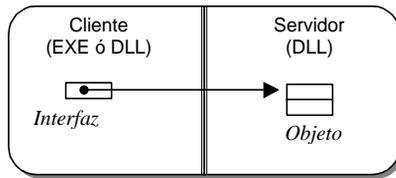
En este capítulo vamos a implementar un *copy hook*, porque aunque es la extensión más sencilla al Explorador, nos permitirá ilustrar todas las fases necesarias para la creación de un servidor COM. El objeto se activará cada vez que intentemos eliminar un directorio que contenga ficheros con extensión *cpp*, y nos preguntará si estamos en nuestros cabales antes de permitir tal atrocidad.

Dentro del proceso, en la misma máquina, remoto...

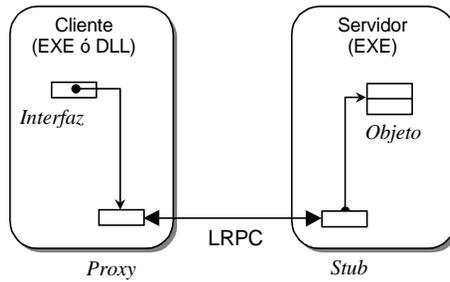
¿Dónde pueden residir los objetos COM? Existen tres modelos diferentes de servidor:

- El objeto reside en una DLL.
- El objeto reside en una aplicación local, dentro del mismo ordenador.
- El objeto reside en una aplicación remota, en otro ordenador de la red.

Los servidores de objetos implementados como DLLs reciben el nombre en inglés de *in-process servers*, o servidores dentro del proceso. Los controles ActiveX se crean obligatoriamente mediante servidores de este estilo. La ventaja de los servidores dentro del proceso es que comparten el mismo espacio de direcciones que la aplicación cliente. De esta forma, el puntero a la interfaz que utiliza que cliente apunta a la verdadera interfaz proporcionada por el objeto, y no es necesario realizar traducción alguna de los parámetros en las llamadas a métodos, resultando en tiempos de ejecución muy eficientes.



El siguiente paso son los servidores locales; los programas de Office pertenecen a esta categoría. La aplicación servidora puede ejecutarse frecuentemente por sí misma, además de efectuar su papel de servidor de objetos. En este modelo, cliente y servidor funcionan sobre espacios de direcciones diferentes y, gracias a las características del modo protegido del procesador, las aplicaciones no tienen acceso directo a las direcciones ajenas. Por lo tanto, OLE implementa un protocolo de comunicación entre aplicaciones simplificado basado en el estándar *Remote Procedure Calls*, conocido como *LRPC*.

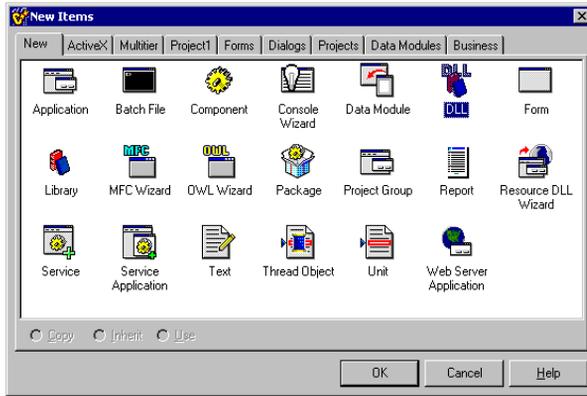


En este caso, el objeto del servidor no puede pasar directamente un puntero a interfaz a sus clientes. Entonces COM crea, de forma transparente para ambos procesos, *objetos delegados (proxies)* en el cliente y en el servidor, que se encargan de la comunicación real entre ambos. El cliente piensa que está trabajando directamente con la interfaz del servidor, pero realmente está actuando sobre una especie de mando de distancia. Cuando se produce una llamada a un método, los parámetros deben empaquetarse en el lado cliente, enviarse vía RPC, para que finalmente el servidor los desempaquete. Lo mismo sucede con los valores retornados por el servidor. A este proceso se le denomina *marshaling*, y más adelante veremos que ciertas interfaces ofrecen soporte predefinido para el mismo.

Si el servidor y el cliente se ejecutan en distintos puestos, el protocolo simplificado se sustituye por el verdadero protocolo de red *RPC*. *RPC* permite ejecutar procedimientos remotos pasando por alto incluso las diferencias en la representación de tipos de datos entre distintos tipos de procesadores. Por ejemplo, el orden entre el byte más significativo y el menos significativo es diferente en los procesadores de Intel y de Motorola. Por supuesto, el *marshaling* debe ocuparse de estos detalles.

La implementación actual de DCOM solamente admite aplicaciones como contenedores remotos de objetos COM, no DLLs. Sin embargo, podemos utilizar alguna aplicación en el servidor que nos sirva de enlace con la DLL. Precisamente, *Microsoft Transaction Server* es una aplicación de este tipo.

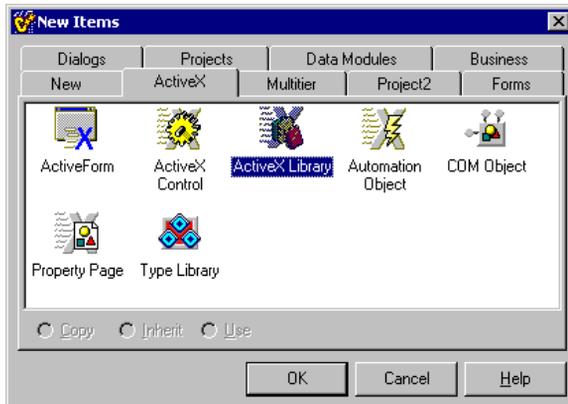
Finalmente, ¿dónde ubicaremos nuestro servidor? Evidentemente, tiene que ser en un proceso local. ¿DLL o ejecutable? Utilizaremos una DLL, para lograr mejores tiempos de carga. Por lo tanto, vamos a la primera página del Depósito de Objetos, y seleccionamos el icono *DLL*:



Este asistente genera una DLL, que guardamos con el nombre *VerifyCpp*. El código generado es el siguiente, después de haber eliminado algunos comentarios e incluir un par de ficheros de cabecera:

```
//-----
#include <vcl.h>
#include <dir.h>
#include <shlobj.h>
#pragma hdrstop
//-----
int WINAPI DllEntryPoint(HINSTANCE hinst, unsigned long reason, void*)
{
    return 1;
}
```

El modo habitual de programar un servidor dentro del proceso, sin embargo, pasa por crear, mediante el Depósito de Objetos, una *ActiveX Library*:



Una DLL preparada para contener servidores COM debe exportar cuatro funciones:

Función	Propósito
<i>DllRegisterServer</i>	Crear entradas de registro del servidor
<i>DllUnregisterServer</i>	Eliminar las entradas del registro
<i>DllGetClassObject</i>	Localización y creación de clases y objetos
<i>DllCanUnloadNow</i>	Decide si puede descargarse la DLL de memoria

La plantilla *ActiveX Library* del Depósito de Objetos crea una DLL con estas funciones, pero basa su implementación en la infame ATL.

Carga y descarga de la DLL

Antes de seguir adelante, vamos a crear una clase auxiliar con una sola instancia, que nos servirá para simplificar un par de tareas relacionadas con la DLL que sirve de anfitriona a nuestra futura clase:

```
class TModule
{
private:
    LONG References;
    HINSTANCE FInstance;
    char FName[MAX_PATH];
    char *GetName() { return FName; }
    void SetInstance(HINSTANCE hinst)
    {
        GetModuleFileName(hinst, FName, MAX_PATH);
    }
public:
    TModule() : References(0) {}
    void __fastcall Lock()
    {
        InterlockedIncrement(&References);
    }
    void __fastcall UnLock()
    {
        InterlockedDecrement(&References);
    }
    HRESULT __fastcall CanUnload()
    {
        return References == 0 ? S_OK : S_FALSE;
    }
    __property HINSTANCE Instance =
        { read = FInstance, write = SetInstance };
    __property char *Name =
        { read = GetName };
} Module;
```

Por una parte, la clase *TModule* define las funciones *Lock* y *UnLock*, que mantienen un contador de referencias al módulo. He definido también una función *CanUnload* que indica si el número de referencias es cero, para poder desactivar el módulo. ¿Ha observado las funciones que se utilizan para incrementar y decrementar el contador? Se trata de funciones del API de Windows que garantizan la consistencia del contador

en entornos potencialmente concurrentes. Tome nota de estas funciones porque las volveremos a ver más adelante.

Además, he incluido un par de propiedades, *Instance* y *Name*, que devuelven respectivamente el número de instancia de la DLL y el nombre del fichero, incluyendo el directorio. La inicialización del número de instancia, y en consecuencia del nombre del fichero base, se realiza en el punto de entrada de la DLL:

```
int WINAPI DllEntryPoint(HINSTANCE hinst, unsigned long reason,
    void*)
{
    Module.Instance = hinst;
    return 1;
}
```

El objetivo final de toda esta parafernalia es implementar una función con el nombre predefinido *DllCanUnloadNow* y exportarla para que pueda ser ejecutada desde fuera de la DLL:

```
STDAPI __export DllCanUnloadNow()
{
    return Module.CanUnload();
}
```

Esta es una de las cuatro funciones que debe implementar toda DLL decente que intente albergar un servidor COM. Así que podemos hacer como el Conde de Montecristo, y tachar a uno de nuestros enemigos de la lista.

OLE y el registro de Windows

Para que las aplicaciones clientes puedan sacar provecho de los servidores COM, estos últimos deben anunciarse como tales en algún lugar. El tablón de anuncios de Windows es el Registro de Configuraciones. Los datos acerca de servidores COM se guardan bajo la clave *HKEY_CLASSES_ROOT*. Estas son, por ejemplo, las entradas del registro asociadas con un servidor COM programado con C++ Builder; el formato es el de los ficheros de exportación del Editor del Registro:

```
[HKEY_CLASSES_ROOT\SemServer.Semaforo]
@="SemaforoObject"
[HKEY_CLASSES_ROOT\SemServer.Semaforo\Clsid]
@="{9346D962-195A-11D1-9412-00A024562074}"
[HKEY_CLASSES_ROOT\CLSID\{9346D962-195A-11D1-9412-00A024562074}]
@="SemaforoObject"
[HKEY_CLASSES_ROOT\CLSID\{9346D962-195A-11D1-9412-00A024562074}\LocalServer32]
@="C:\\MARTEENS\\PROGS\\SEMSERVER\\SEMSERVER.EXE"
[HKEY_CLASSES_ROOT\CLSID\{9346D962-195A-11D1-9412-00A024562074}\ProgID]
@="SemServer.Semaforo"
```

```
[HKEY_CLASSES_ROOT\CLSID\{9346D962-195A-11D1-9412-00A024562074}\Version]
@="1.0"
[HKEY_CLASSES_ROOT\CLSID\{9346D962-195A-11D1-9412-00A024562074}\TypeLib]
@="{9346D960-195A-11D1-9412-00A024562074}"
```

Para explicar el significado de esta información, lo mejor es mostrar paso a paso lo que ocurre cuando una aplicación pide la creación de un objeto COM. Es más fácil comenzar con una llamada sencilla a *CoCreateInstance*:

```
IUnknown *intf;
CoCreateInstance(CLSID_ISemaforo, NULL, CLSCTX_ALL,
    IID_IUnknown, (void**) &intf);
```

- El identificador de clase se busca en la clave *CLSID* de la raíz de las clases del registro: *HKEY_CLASSES_ROOT*.
- Una vez encontrada la clave, se busca una subclave dentro de la misma que nos indique de qué tipo de servidor se trata. Los tipos posibles son: *InprocServer32*, para servidores dentro del proceso, y *LocalServer32*, para servidores locales implementados como ejecutables.
- La clave encontrada contiene como valor predeterminado la ruta a la DLL o al ejecutable que actuará como servidor. Aquí nos detendremos, de momento.

La mayoría de los desarrolladores que se inician en el mundo de la programación COM perciben como una tragedia el tener que trabajar con identificadores globales de clase. No veo por qué. Cuando usted tenga que crear un objeto de una clase, sencillamente tendrá a su disposición un fichero de cabecera de C que contendrá las declaraciones necesarias, incluyendo los identificadores de clase y de interfaz. En caso contrario, tendrá todas las facilidades para generar dicho fichero; se lo prometo, pero tendrá que esperar un poco, hasta ver las bibliotecas de tipo, para comprender el motivo.

No obstante, reconozco que en ocasiones puede ser más cómodo disponer de una representación “legible” del nombre de una clase, que se traduzca unívocamente a un CLSID. En la jerga de COM, a estos nombres se les conoce como *identificadores de programas*, ó *programmatic identifiers*. Las siguientes funciones convierten un identificador de programa en un identificador de clase, y viceversa:

```
HRESULT CLSIDFromProgID(const OLECHAR *pwszProgID, CLSID *pclsid);
HRESULT ProgIDFromCLSID(REFCLSID rclsid, OLECHAR **pwszProgID);
// ¿Sabéis una cosa? ¡Odio la notación húngara!
```

Para que la conversión sea eficiente en las dos direcciones, se deben almacenar en el registro entradas redundantes. Directamente dentro de *HKEY_CLASSES_ROOT* se almacena una clave con el identificador de programa; casi siempre, éste tiene el formato *Nombre.Servidor.NombreClase*. Dentro de dicha clave, existe una subclave *Clsid*,

que naturalmente contiene el identificador de clase correspondiente. Por su parte, la clave que explicamos primero contiene una subclave *ProgId* que nos indica el identificador de programa. Circuito cerrado y asunto concluido.

Bueno, no tan rápido. La siguiente complicación surge por la existencia de identificadores de programa dependientes e independientes de la versión. Como ejercicio, si tiene Word instalado en su máquina, utilice el Editor del Registro para explorar toda la basura que graba este procesador de textos en su máquina.

Registrando el servidor

Existen técnicas estándar para registrar un servidor COM, y dependen de si el servidor es una DLL o un ejecutable. En el primer caso, la DLL debe implementar y exportar las siguientes dos funciones:

```
STDAPI __export DllRegisterServer(void);
STDAPI __export DllUnregisterServer(void);
```

Si se trata de un ejecutable, el convenio para registrarlo es invocarlo con el parámetro */regserver*, o sin parámetros. En el primer caso, solamente se graban las entradas del registro y la ejecución del programa termina inmediatamente. Para borrar las entradas del registro, el convenio consiste en utilizar el parámetro */unregserver*.

Inprise ofrece el programa *regsvr.exe*, cuyo código fuente se encuentra entre los programas de demostración de C++ Builder, para registrar indistintamente servidores ejecutables y bibliotecas dinámicas.

Necesitamos un GUID que identifique a la clase que vamos a implementar. Para obtenerlo, colocamos el cursor en cualquier línea vacía del editor de código y pulsamos la combinación de teclas CTRL+MAY+G, para obtener la siguiente cadena:

```
{ ' {26042460-DF02-11D2-B67D-0000E8D7F7B2} ' }
```

Tenemos que maquillar la cadena obtenida para que sea utilizable por C++ Builder:

```
GUID CLSID_VerifyCpp = {0x26042460L, 0xDF02, 0x11D2,
    {0xB6, 0x7D, 0x00, 0x00, 0xE8, 0xD7, 0xF7, 0xB2}};
```

Antes de programar a *DllRegisterServer* y su compañera, voy a definir una función auxiliar para crear más fácilmente las entradas en el registro:

```
void __fastcall CreateStrKey(void *Raiz, const char *Clave,
    const char *Param, const char *Datos)
```

```

{
    HKEY hkey;
    long error = RegCreateKey(Raiz, Clave, &hkey);
    if (error == ERROR_SUCCESS)
    {
        error = RegSetValueEx(hkey, Param, 0, REG_SZ,
            (const unsigned char*) Datos, strlen(Datos) + 1);
        RegCloseKey(hkey);
    }
    if (error != ERROR_SUCCESS)
        throw Exception("Error creando clave");
}

```

Ya es fácil implementar las dos funciones que tocan el Registro; primero la que lo limpia, y después la que lo ensucia:

```

STDAPI __export DllUnregisterServer()
{
    HRESULT Rslt = S_OK;
    if (RegDeleteKey(HKEY_CLASSES_ROOT,
        "Directory\\shelllex\\CopyHookHandlers\\VerifyCpp")
        != ERROR_SUCCESS) Rslt = S_FALSE;
    if (RegDeleteKey(HKEY_CLASSES_ROOT,
        "CLSID\\{26042460-DF02-11D2-B67D-0000E8D7F7B2}"
        "\\InprocServer32") != ERROR_SUCCESS) Rslt = S_FALSE;
    if (RegDeleteKey(HKEY_CLASSES_ROOT,
        "CLSID\\{26042460-DF02-11D2-B67D-0000E8D7F7B2}"
        != ERROR_SUCCESS) Rslt = S_FALSE;
    return Rslt;
}

STDAPI __export DllRegisterServer()
{
    try
    {
        CreateStrKey(HKEY_CLASSES_ROOT,
            "CLSID\\{26042460-DF02-11D2-B67D-0000E8D7F7B2}",
            0, "VerifyCpp");
        CreateStrKey(HKEY_CLASSES_ROOT,
            "CLSID\\{26042460-DF02-11D2-B67D-0000E8D7F7B2}"
            "\\InprocServer32", 0, Module.Name);
        CreateStrKey(HKEY_CLASSES_ROOT,
            "CLSID\\{26042460-DF02-11D2-B67D-0000E8D7F7B2}"
            "\\InprocServer32", "ThreadingModel", "Apartment");
        CreateStrKey(HKEY_CLASSES_ROOT,
            "Directory\\shelllex\\CopyHookHandlers\\VerifyCpp",
            0, "{26042460-DF02-11D2-B67D-0000E8D7F7B2}");
    }
    catch(...)
    {
        DllUnregisterServer();
        return SELFREG_E_CLASS;
    }
    return S_OK;
}

```

Es fácil reconocer la grabación de las entradas que he explicado en la sección anterior. Aquí, sin embargo, no estoy grabando información para que los clientes utilicen

identificadores de programa. Para las extensiones del Explorador, además, hace falta añadir claves adicionales, pues el Explorador no va a revisar todas las clases registradas en el sistema, crear instancias de ellas y determinar cuál implementa una extensión y cuál no. En el caso de los *copy books*, debemos añadir valores en las siguientes claves:

```
HKEY_CLASSES_ROOT\Directory\shellex\CopyHookHandlers
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\
Shell Extensions\Approved
```

La segunda clave es necesaria únicamente cuando se quiere registrar la extensión en Windows NT. Esto se lo dejo como ejercicio.

La función *DllRegisterServer* asigna al parámetro *ThreadingModel* el valor *Apartment*. Este tiene que ver con el modelo de concurrencia permitido por el servidor, novedad introducida por la versión COM para objetos remotos (DCOM). Los modelos de concurrencia serán estudiados en los próximos capítulos.

Implementación de interfaces

Ahora veremos cómo se crea la clase, en el sentido tradicional del término, que implementará los métodos ofrecidos por la interfaz *ICopyHook*, que no es otra cosa para C++ que una clase abstracta.

```
class TVerifyCpp : public ICopyHook
{
private:
    LONG Referencias;
public:
    TVerifyCpp();
    STDMETHODIMP QueryInterface(REFIID, LPVOID*);
    STDMETHODIMP_(ULONG) AddRef();
    STDMETHODIMP_(ULONG) Release();
    STDMETHODIMP_(UINT) CopyCallback(HWND, UINT wFunc, UINT,
        LPCSTR pszSrcFile, DWORD, LPCSTR, DWORD);
};
```

La clase *TVerifyCpp* hereda de la clase *ICopyHook*, definida en *shlobj*, y que declara los métodos de la interfaz correspondiente como funciones miembros virtuales puras. Por lo tanto, si queremos poder crear instancias de *TVerifyCpp* tendremos que proporcionar un cuerpo a cada uno de estos métodos. Los tres métodos que siguen al constructor son nuestros viejos conocidos soportados por *IUnknown*.

Se supone que *CopyCallback*, el método específico de *ICopyHook*, es llamado por Windows cuando vamos a realizar alguna operación sobre un directorio. *wFunc* es la

operación que se va a ejecutar, y *pszSrcFile* es el nombre del directorio origen de la operación. Esta función puede devolver uno de los tres valores siguientes:

Valor de retorno	Significado
<i>IDYES</i>	Se aprueba la operación
<i>IDNO</i>	Cancela solamente esta operación
<i>IDCANCEL</i>	Cancela esta operación y las siguientes

Primero nos ocuparemos de contar referencias, implementando los métodos *AddRef* y *Release*:

```
TVerifyCpp::TVerifyCpp() : Referencias(0) {}

STDMETHODIMP_(ULONG) TVerifyCpp::AddRef()
{
    Module.Lock();
    return InterlockedIncrement(&Referencias);
}

STDMETHODIMP_(ULONG) TVerifyCpp::Release()
{
    Module.UnLock();
    LONG r = InterlockedDecrement(&Referencias);
    if (r == 0)
        delete this;
    return r;
}
```

He vuelto a utilizar a *InterlockedIncrement* y a su pareja para mantener un contador con las referencias a un objeto de la clase. Además, es necesario manejar también las referencias al módulo, para lo cual se utilizan los métodos *Lock* y *UnLock* definidos para la variable global *Module*. Observe en *Release* cómo se libera la memoria del objeto cuando las referencias al mismo caen a cero. Quiere decir que todas las instancias de la clase *TVerifyCpp* deben crearse en memoria dinámica.

Para completar la implementación de los métodos de *IUnknown* debemos dar un cuerpo a *QueryInterface*:

```
STDMETHODIMP TVerifyCpp::QueryInterface(REFIID riid, LPVOID* ppv)
{
    if (riid == IID_IUnknown || riid == IID_IShellCopyHook)
        *ppv = this;
    else
        return E_NOINTERFACE;
    AddRef();
    return S_OK;
}
```

Nuestra clase implementa la interfaz identificada mediante *IID_IShellCopyHook*, definida por Windows. Pero también implementa explícitamente la interfaz *IUnknown*, aunque está implícitamente incluida dentro de *ICopyHook*. Cuando se pide cualquiera

de estas dos interfaces, se devuelve directamente un puntero al propio objeto. Si la clase tuviera que implementar más de una interfaz explícita, tendríamos que utilizar conversiones dinámicas (*dynamic cast*) para devolver el puntero adecuado dentro del objeto.

MUY IMPORTANTE

Observe cómo se llama a *AddRef* cuando se devuelve sin problemas un puntero a interfaz. Esta es una de las reglas del juego que comentaba en el capítulo anterior: el cliente no tiene que llamar a *AddRef* después de obtener una interfaz mediante *QueryInterface*, porque este método ya se encarga de ello. Durante la escritura del presente capítulo, se me olvidó ese pequeño detalle, y el ejemplo fallaba de forma ignominiosa. Después de consumir litros de café y toneladas de aspirina, logré darme cuenta de la omisión, pero me quedé preocupado por mi salud gástrica. Han pasado un par de meses desde entonces y aún estoy vivo.

Finalmente, ésta es la implementación del método *CopyCallback* declarado en la interfaz *ICopyHook*:

```
STDMETHODIMP_(UINT) CopyCallback(HWND hwnd, UINT wFunc, UINT wFlags,
    LPCSTR pszSrcFile, DWORD dwSrcAttribs, LPCSTR pszDestFile,
    DWORD dwDestAttribs)
{
    if (wFunc == FO_DELETE)
    {
        struct ffbk ffbk;
        char fichero[256];
        strcpy(fichero, pszSrcFile);
        strcat(fichero, "\\*.cpp");
        int done = findfirst(fichero, &ffbkl, 0);
        findclose(&ffbkl);
        if (done == 0)
            return MessageBox(hwnd, "¿Me cepillo este directorio?",
                ";Precaución!", MB_YESNOCANCEL);
    }
    return IDYES;
}
```

El huevo, la gallina y las fábricas de clases

Una de las interfaces importantes definidas por COM es *IClassFactory*, que proporciona los servicios necesarios para la creación de objetos. De esta manera COM resuelve el eterno dilema de quién fue primero, si el huevo, la gallina o la tortilla de patatas: si usted quiere crear un objeto, la aplicación o DLL servidora debe ser capaz de ofrecer primero una interfaz *IClassFactory* para que sea ésta quien se encargue de la creación. Las clases que soportan esta interfaz reciben el nombre de *fábricas de clases*.

¿Para qué se necesitan fábricas de clases? Piense por un momento en una clase de C++ “clásica”, y en la forma que se llama al constructor en el caso de los objetos creados en la memoria dinámica. De cierto modo, se puede ver al constructor como un método especial estático (**static**) o de clase; de hecho, la notación utilizada por un lenguaje como Delphi refuerza esta percepción. ¿Tenemos constructores o métodos estáticos en COM? Por simplicidad del modelo, no existe tal posibilidad actualmente. Sin embargo, para simular estos recursos puede declararse una clase auxiliar, de la que siempre se creará una sola instancia, y utilizar los métodos de esta instancia como métodos globales de la clase principal, que se pueden ejecutar existan o no objetos de esta clase.

¿Hay algo que nos obligue a utilizar fábricas de clase como método de construcción? Bueno, no es necesario que estos “objetos de clase” se implementen como derivados de *IClassFactory*. Pudiéramos utilizar nuestra propia interfaz personal para la instanciación ... siempre y cuando advirtiésemos a los clientes del cambio. Ahora bien, si existe un estándar, ¿para qué inventarnos otro? Además, ciertas aplicaciones COM exigen que los objetos que manejan se creen mediante *IClassFactory*; una de ellas es *Microsoft Transaction Server*.

Como todas las interfaces, *IClassFactory* contiene los métodos de *IUnknown*, además de definir sus métodos específicos:

```
HRESULT CreateInstance(
    IUnknown *pUnkOuter, REFIID riid, LPVOID *ppv);
HRESULT LockServer(BOOL fLock);
```

CreateInstance hace el papel de constructor de objetos: se le suministra el identificador único de clase y, si puede crear objetos de esa clase, devuelve un puntero a la interfaz *IUnknown* del objeto recién creado. *LockServer*, por su parte, mantiene a la aplicación servidora en memoria, para acelerar la creación posterior de objetos por la misma.

La idea es que todo servidor COM debe crear una instancia de un objeto que implemente la interfaz *IClassFactory*, y ponerlo a disposición del sistema operativo. En el caso de un ejecutable, se debe llamar a la función *CoRegisterClassObject* por cada fábrica de clase soportada. En el caso de una DLL, se debe exportar la siguiente función, para que sea ejecutada por el sistema:

```
STDAPI __export DllGetClassObject(REFCLSID rclsid, REFIID riid,
    LPVOID* ppv);
```

Ya estamos en condiciones de terminar la descripción del proceso de creación de un objeto componente. Al principio del capítulo habíamos esbozado los primeros pasos, que se pueden resumir del siguiente modo:

- Dado un identificador de clase o de programa, el sistema operativo se las apaña con el Registro para encontrar si la clase reside en un EXE o en una DLL, y ejecuta el correspondiente servidor.

A continuación:

- Se le solicita al servidor una fábrica de clase para la clase que queremos instanciar. Si el servidor es una DLL, el sistema llama a su función *DllGetClassObject*. Si es un EXE, se busca dentro de la lista de fábricas de clase registradas.
- Una vez localizado el objeto, se llama a su método *CreateInstance* para generar el objeto deseado.

Conociendo el algoritmo anterior podemos plantearnos optimizaciones al proceso de creación de un objeto. Si lo que deseamos es crear varios objetos de una misma clase, por ejemplo, en vez de llamar a *CoCreateInstance* varias veces tenemos la posibilidad de llamar una sola vez a la función *CoGetClassObject*, y una vez que tenemos un puntero a la interfaz *IClassFactory*, llamamos cuantas veces sea necesario a *CreateInstance*.

Implementando la fábrica de clases

Para terminar con nuestro servidor, definiremos una clase que implemente la interfaz *IClassFactory*:

```
class TVerifyCppCF : public IClassFactory
{
public:
    STDMETHODCALLTYPE AddRef()
    {
        Module.Lock();
        return 2;
    }
    STDMETHODCALLTYPE Release()
    {
        Module.Unlock();
        return 1;
    }
    STDMETHODCALLTYPE QueryInterface(REFIID, LPVOID*);
    STDMETHODCALLTYPE CreateInstance(
        IUnknown *pUnkOuter, REFIID riid, LPVOID *ppv);
    STDMETHODCALLTYPE LockServer(BOOL fLock);
};
```

¡Un momento! ¿Por qué esta implementación tan extraña para *AddRef* y para *Release*? Es que las instancias de esta clase van a residir en memoria estática, y no nos interesa que *Release* destruya el objeto. Por lo tanto, tras cualquier llamada a *Release*, nuestro contador de referencias valdrá siempre *1*. Eso sí, nos debemos ocupar del contador de referencias del módulo.

El método *QueryInterface* se implementa con igual facilidad:

```
STDMETHODIMP TVerifyCppCF::QueryInterface(REFIID riid, LPVOID* ppv)
{
    if (riid == IID_IUnknown || riid == IID_IClassFactory)
        *ppv = this;
    else
        return E_NOINTERFACE;
    AddRef();
    return S_OK;
}
```

Pero el método más importante es *CreateInstance*. Para simplificar, no soportaremos la creación de este objeto como parte de un agregado. Y debemos acordarnos de incrementar el contador de referencias del objeto recién creado:

```
STDMETHODIMP TVerifyCppCF::CreateInstance(
    IUnknown *pUnkOuter, REFIID riid, LPVOID *ppv)
{
    *ppv = 0;
    if (pUnkOuter != 0)
        return CLASS_E_NOAGGREGATION;
    TVerifyCpp *vcpp = new TVerifyCpp;
    if (vcpp == 0)
        return E_OUTOFMEMORY;
    vcpp->AddRef();
    *ppv = LPVOID(vcpp);
    return S_OK;
}
```

En la implementación de *LockServer* nos ocupamos de llamar a los métodos *Lock* y *UnLock* del objeto módulo que creamos al principio del capítulo. *LockServer* es utilizado por los clientes cuando, por alguna razón, les interesa mantener el servidor en memoria:

```
STDMETHODIMP TVerifyCppCF::LockServer(BOOL fLock)
{
    if (fLock)
        Module.Lock();
    else
        Module.UnLock();
    return S_OK;
}
```

Finalmente, he aquí la implementación de la función *DllGetClassObject*:

```
STDAPI __export DllGetClassObject(REFCLSID rclsid, REFIID riid,
    void **ppv)
{
    static TVerifyCppCF verifyCppCF;
    if (rclsid == CLSID_VerifyCpp)
        return verifyCppCF.QueryInterface(riid, ppv);
    *ppv = 0;
}
```

```
        return CLASS_E_CLASSNOTAVAILABLE;  
    }
```

Como nuestro servidor solamente implementa una clase COM, debe reaccionar únicamente cuando el parámetro *rvlsid* trae el GUID de dicha clase. La fábrica de clases se crea como un objeto estático, y la petición del objeto de clase se satisface llamando a *QueryInterface*; de este modo, se llama indirectamente a *AddRef* sobre el dicho objeto.

Los servidores situados en ejecutables deben crear un objeto de la fábrica de clases, y registrarlo llamando a la función *CoRegisterClassObject*, al comienzo de la ejecución del programa.

Automatización OLE: controladores

ES EN ESTE PUNTO DONDE LOS MANUALES DE C++ Builder realmente comienzan sus explicaciones. Se denomina *Automatización OLE* al área de COM que permite manejar métodos de objetos mediante una interfaz de macros. Veremos cómo esta técnica se amplía y mejora mediante el uso de *interfaces duales*. El plan de trabajo es sencillo. En este primer capítulo actuaremos como clientes, o como dice la jerga, *controladores* de automatización. En el próximo capítulo aprenderemos a programar los servidores.

¿Por qué existe la Automatización OLE?

Quizás la interfaz más popular de COM sea *IDispatch*, que define los servicios de *automatización OLE* (*OLE Automation*). Esta es una forma de ejecutar métodos de un objeto COM, que puede residir en una DLL o en otra aplicación. La llamada a los métodos remotos puede efectuarse por medio del nombre de los mismos, en el estilo de un lenguaje de macros. De hecho, la automatización OLE sustituye al viejo mecanismo de ejecución de macros de DDE. La automatización OLE se ha hecho popular gracias sobre todo a que Word, y los restantes productos de Office, pueden actuar como servidores de automatización. Pero muchos otros productos comienzan a aprovechar esta técnica, ya sea como servidores o controladores. Por ejemplo, MS SQL Server puede controlar objetos de automatización desde *scripts* programados en Transact-SQL. La automatización también es importante para C++ Builder, pues es una de las formas en que se implementan las técnicas de acceso a bases de datos remotas con Midas. C++ Builder define una interfaz *IDataBroker*, derivada de *IDispatch*, y por medio de la misma comunica las aplicaciones clientes con los servidores de aplicaciones.

He dicho que la automatización OLE está basada en macros. Pero, ¿no es obsoleto y peligroso recurrir a macros para el control de objetos? Por supuesto que sí: creo que a estas alturas a nadie se le ocurriría defender la falta de comprobación estática de tipos, excepto a los adictos a Visual Basic. Y fue principalmente por culpa de Visual

Basic que Microsoft diseñó *IDispatch*. Colateralmente, sin embargo, la automatización OLE ofrece algunos beneficios, siendo el principal la implementación implícita del *marshaling*: la organización de los datos para su transferencia entre procesos. Además, y como veremos más adelante, el uso de interfaces duales permite evitar muchos de los riesgos de esta técnica.

No es que yo tenga inclinaciones sádicas, pero es mi obligación mostrarle la declaración de *IDispatch* en IDL:

```
[object, uuid(00020400-0000-0000-C000-000000000046)]
interface IDispatch : IUnknown
{
    HRESULT GetTypeInfoCount([out] UINT *pctinfo);
    HRESULT GetTypeInfo(
        [in] UINT iTInfo,
        [in] LCID lcid,
        [out] ITypeInfo **ppTInfo);
    HRESULT GetIDsOfNames(
        [in] REFIID riid,
        [in, size_is(cNames)] LPOLESTR *rgszNames,
        [in] UINT cNames,
        [in] LCID lcid,
        [out, size_is(cNames)] DISPID *rgid);
    HRESULT Invoke(
        [in] DISPID id,
        [in] REFIID riid,
        [in] LCID lcid,
        [in] WORD wFlags,
        [in, out] DISPPARAMS *pDispParams,
        [out] VARIANT *pVarResult,
        [out] EXCEPINFO *pExcepInfo,
        [out] UINT *puArgErr);
}
```

No se preocupe, yo mismo soy incapaz de recordar todos los parámetros de estas funciones. En realidad, nunca he utilizado directamente la interfaz *IDispatch*. Dada su complejidad, la mayoría de los lenguajes ofrece algún tipo de encapsulamiento para realizar llamadas a la misma. En Delphi y en Visual Basic, ese encapsulamiento nos puede llegar a engañar, de modo que no nos demos cuenta de que realmente estamos utilizando macros. En C++ existen clases para esta tarea, que estudiaremos en breve.

El núcleo de *IDispatch* es el método *Invoke*, que sirve para ejecutar una macro en el servidor. Para tal propósito, *Invoke* permite la especificación de parámetros de entrada y salida, de valores de retorno para funciones y la propagación de excepciones desde el servidor al cliente. Ahora bien, *Invoke* no utiliza cadenas de caracteres para especificar el método a ejecutar. Por el contrario, el método se identifica mediante un valor numérico, que se pasa en el parámetro *id*, de tipo *DISPID*. Lo mismo sucede con los nombres de parámetros. *Invoke* admite pasar parámetros por nombre a los métodos que activa, lo cual evita que tengamos que recordar siempre la posición exacta de un parámetro cuando la macro tiene muchos.

La función *GetIDsOfNames* es la encargada de traducir los nombres de métodos y de parámetros en códigos numéricos, para pasarlos posteriormente a *Invoke*. ¿Por qué la ejecución de una macro se realiza en estos dos pasos, traducción a identificador numérico y posterior ejecución? La razón es evidente: la lucha por la eficiencia. Si vamos a ejecutar un método mediante *Invoke* varias veces consecutivas, quizás dentro de un bucle, no queremos que en cada ejecución el objeto tenga que efectuar una larga serie de comparaciones de cadenas de caracteres; más rápido es comparar dos enteros.

¿Cómo se asignan estos valores enteros a los métodos de automatización? En el lenguaje IDL se utiliza un tipo especial de declaración: las interfaces de despacho, o *dispatch interfaces*:

```
[ uuid(20D56981-3BA7-11D2-837B-0000E8D7F7B2) ]
dispinterface IGenericReport
{
methods:
[id(1)] void Print([in] BOOL DoSetup);
[id(2)] void Preview();
}
```

Esta no es una interfaz, en el sentido normal de la palabra, sino que es una tabla para el consumo interno del objeto que implemente una interfaz de automatización con los dos métodos anteriores.

Controladores de automatización con variantes

A los clientes de objetos de automatización se les llama *controladores de automatización*. Un controlador de automatización escrito en C++ Builder puede obtener un puntero a la interfaz *IDispatch* de un objeto de automatización y almacenarlo en una variable de tipo *Variant* mediante una llamada a la función estática *CreateObject*, de la propia clase *Variant*:

```
static Variant __fastcall CreateObject(const String& ProgID);
```

Esta función utiliza el identificador de programa de la clase, en vez del identificador de clase. Una vez que tenemos la interfaz *IDispatch* bien oculta dentro del variante, podemos utilizar el método *Exec* para ejecutar las macros que deseemos:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
Variant word = Variant::CreateObject("Word.Basic");
word.Exec(Procedure("AppShow"));
word.Exec(Procedure("FileNewDefault"));
for (int i = 0; i < Mem1->Lines->Count; i++)
word.Exec(Procedure("Insert") <<
(Mem1->Lines->Strings[i] + "\n"));
}
```

```

word.Exec(Procedure("FileSaveAs") <<
    ChangeFileExt(Application->ExeName, ".doc");
}

```

El ejemplo anterior es un clásico de los libros de C++ Builder. Hemos obtenido un puntero a un objeto de la clase *Word.Basic*, exportada por Microsoft Word. Los métodos que se llaman a continuación son métodos exportados por esa clase. El primero, *AppShow*, hace visible a Word, el segundo, *FileNewDefault*, crea un fichero con propiedades por omisión, *Insert* inserta una línea y *FileSaveAs* guarda el fichero en el directorio de la aplicación. No hay que destruir explícitamente al objeto creado, pues el compilador se encarga de hacerlo llamando al destructor de *Variant* cuando desaparece la variable local *word*. Si quisiéramos destruir el objeto explícitamente, tendríamos que llamar al método *Clear*, que asigna el valor especial *Unassigned*:

```

word.Clear();
// Equivalente a:
word = Unassigned;

```

Para especificar los nombres de métodos o macros se utilizan las clases *Procedure* y *Function*. En el ejemplo anterior los objetos de estas clases se crean *inline*, de forma anónima, pero también pueden crearse explícitamente, sobre todo en los casos en que buscamos mayor eficiencia. Por ejemplo:

```

Procedure insert("Insert");
for (int i = 0; i < Memol->Lines->Count; i++)
    word.Exec(insert << (Memol->Lines->Strings[i] + "\n"));

```

Propiedades OLE y parámetros por nombre

También puede controlarse Word mediante la clase *Word.Application*, a partir de Office 97. Al parecer, *Word.Basic* está destinada a desaparecer en próximas versiones del producto. En vez de definir un sinnúmero de métodos en una sola y monstruosa clase, con las nuevas clases se crea una jerarquía de objetos: un objeto de clase *Application* contiene un puntero a una colección *Documents*, que contiene objetos de tipo *Document*, etcétera, etcétera. El siguiente ejemplo es equivalente al anterior:

```

void __fastcall TForm1::Button2Click(TObject *Sender)
{
    Variant WApp = Variant::CreateObject("Word.Application");
    WApp.OlePropertySet("Visible", True);
    WApp.OlePropertyGet("Documents").OleFunction("Add");
    Variant WDoc = WApp.OlePropertyGet("ActiveDocument");
    WDoc.OlePropertyGet("Content").Exec(
        Procedure("InsertAfter") << "Mira lo que hago con Word\n");
    WDoc.Exec(Procedure("SaveAs") << NamedParm(
        "FileName", ChangeFileExt(Application->ExeName, ".doc")));
}

```

Aquí hay novedades. Las interfaces *dispatch* ofrecen soporte para propiedades, que internamente se implementan mediante métodos de lectura y escritura. Los métodos *OleGetProperty* y *OleSetProperty* sirven para leer y modificar dichas propiedades:

```
WApp.OlePropertySet("Visible", True);
WApp.OlePropertyGet("Documents").OleFunction("Add");
```

En la última instrucción del ejemplo he mostrado un ejemplo de cómo utilizar un parámetro con nombre:

```
WDoc.Exec(Procedure("SaveAs") << NamedParm(
    "FileName", ChangeFileExt(Application->ExeName, ".doc")));
```

Resulta que el método *SaveAs*, del objeto *Document* de Word, tiene nada más y nada menos que once parámetros. De todos ellos, solamente nos importa el primero, pues los restantes van a utilizar valores por omisión. Podíamos haber escrito esa instrucción de esta otra forma, aprovechando que el parámetro que suministramos es el primero:

```
WDoc.Exec(Procedure("SaveAs") <<
    ChangeFileExt(Application->ExeName, ".doc"));
```

Recuerde que esta rocambolesca sintaxis se traduce, en definitiva, a llamadas al método *GetIDsOfNames* y en operaciones sobre los parámetros de *Invoke*.

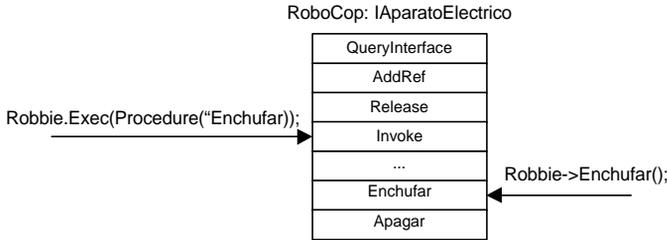
Interfaces duales

A pesar de la inseguridad que provoca la carencia de verificaciones estáticas de tipos, hay que reconocer que en ciertas ocasiones puede ser ventajoso un acoplamiento débil entre cliente y servidor. El cliente no tiene por qué conocer todos los detalles de la interfaz del servidor. Piense un momento en un servidor de automatización como Word, que ofrece cientos de funciones, con montones de parámetros y valores por omisión. A usted, sin embargo, solamente le interesa saber cómo abrir un fichero e imprimirlo. ¿Para qué tener que aprender todos los demás métodos?

Pero también hay ventajas a la inversa: un cliente puede trabajar con más servidores. Si yo necesito ejecutar un método *Imprimir* y sé que determinado servidor lo implementa, me da lo mismo la existencia de otros métodos dentro de ese servidor, siempre que pueda utilizar su interfaz *IDispatch*.

No obstante, y a pesar de las mejoras en tiempo de acceso que ofrecen los *DispId*, las llamadas a métodos por medio de *Invoke* son inevitablemente lentas e inseguras. Para paliar el problema, existen las denominadas *interfaces duales*. Se dice que una clase so-

porta una interfaz dual cuando implementa una interfaz derivada de *IDispatch*, en la cual se han incluido los mismos métodos que pueden ejecutarse vía *Invoke*.



¿Le ocasiona algún trastorno al programador el hecho de que determinada interfaz sea dual? Ninguno, pues puede seguir utilizando los métodos variantes para controlar el método mediante macros, pero también puede trabajar directamente con la interfaz ampliada, del mismo modo en que lo hacíamos en el capítulo inicial sobre COM.

Bibliotecas de tipos

Para que el programador pueda aprovechar una interfaz dual en C++ Builder debe tener a su alcance el fichero de cabecera con las declaraciones de los tipos de interfaz, lo cual no sucedía cuando utilizaba solamente la interfaz *IDispatch*. Antes bastaba con una documentación informal acerca de cuáles macros utilizar y con qué parámetros. La descripción podía incluso estar destinada a programadores de Visual Basic, pero era fácil hacer corresponder estos conceptos a C++, a Java o a Delphi.

Cuando presentamos el ejemplo de creación de accesos directos utilizamos las interfaces *IShellLink* y *IPersistFile*. ¿De dónde las sacamos? Como son interfaces mantenidas directamente por Windows, los ficheros de cabeceras se suministran con el propio compilador de C++, o por el propio Kit de Desarrollo del sistema operativo. ¿Quiere decir esto que cada implementador de una interfaz dual está obligado a suministrar los ficheros de cabecera correspondiente? Aunque no estaría mal, esta idea hace aguas. Recuerde que uno de los objetivos es que nuestros objetos sean aprovechables por cualquier lenguaje medianamente decente²³. No es práctico ni factible prever todas las plataformas de desarrollo desde donde se puede utilizar el servidor.

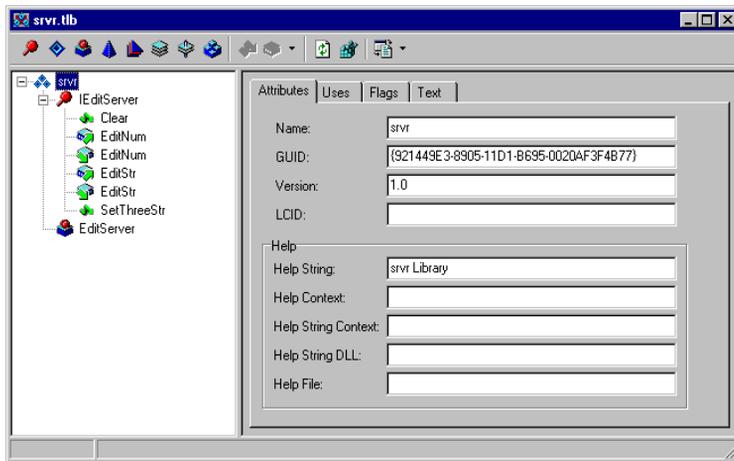
Una solución es utilizar el lenguaje IDL para describir las interfaces. Sería entonces responsabilidad del lenguaje proporcionar alguna herramienta de compilación que traduzca automáticamente las descripciones y genere los ficheros de cabecera correspondientes. Pero aquí también puede aplicarse el encapsulamiento. En vez de proporcionar el fichero ejecutable o la DLL que contiene la clase *más* un fichero IDL

²³ Y también por Visual Basic.

con las declaraciones, ¿por qué no incluimos las declaraciones directamente dentro del servidor?

Ese es el propósito de las *bibliotecas de tipos* (*type libraries*). Una biblioteca de tipos es una representación binaria e independiente de cualquier lenguaje que se obtiene a partir de un fichero IDL. Puede almacenarse en un fichero independiente, casi siempre de extensión *tlb* ó *olb*, o incluirse como un recurso de Windows, de clase *typelib*, dentro de un servidor COM.

Hay ciertos tipos de servidores para los cuales es obligatorio suministrar una biblioteca de tipos, como los controles ActiveX. Pero en general, se trata de un bonito detalle por parte del implementador. En el próximo capítulo veremos cómo C++ Builder permite la edición visual de una biblioteca de tipos cuando creamos servidores de automatización:

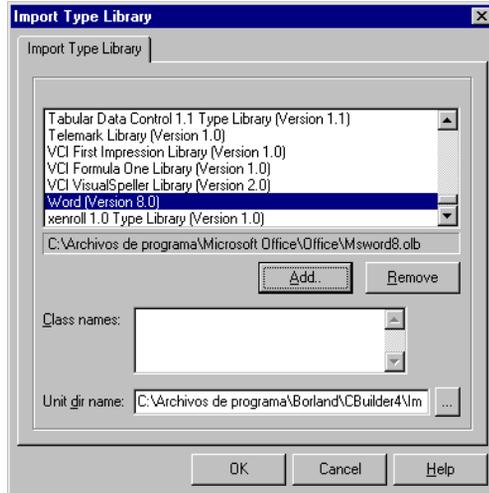


Importación de bibliotecas de tipos

¿Cómo transformamos una biblioteca de tipos en un fichero de cabecera para C++ Builder, de modo que podamos aprovechar una interfaz dual? Necesitamos *importar* la biblioteca, para lo cual C++ Builder ofrece el comando de menú *Project | Import type library*. Cuando se ejecuta dicho comando, aparece en primer lugar la lista de bibliotecas registradas. Con los botones *Add* y *Remove* podemos registrar una nueva biblioteca, o eliminar alguna del Registro. En el caso de que la biblioteca contenga uno o más controles ActiveX, la unidad generada contendrá declaraciones de componentes para los mismos, y un método *Register* para poder incluir estos componentes en la Paleta de Componentes, si así lo desea.

Lo que necesitamos en estos momentos son las interfaces duales para poder controlar a Word. Al ejecutar el comando de importación, tenemos que buscar la biblioteca de tipos de Word (no se encuentra dentro del ejecutable) mediante el botón *Add*. El fichero es el siguiente:

C:\Archivos de programa\Microsoft Office\Office\Msword8.olb



Una vez localizado el fichero, pulse el botón *OK* y póngase cómodo, pues la operación tarda un poco en completarse. Al final de la misma tendremos a nuestra disposición un nuevo par de ficheros: *Word_TLB.cpp* y *Word_TLB.h*, que podemos incluir en nuestros proyectos. Compruebe que, efectivamente, contienen todas las interfaces y declaraciones necesarias para acceder a Word 97.

La traducción literal produce una serie de interfaces de bajo nivel; digo de bajo nivel porque no soportan propiedades, punteros inteligentes y todas esas pequeñas mejoras que hacen soportable trabajar con C++. Por ejemplo, para extraer la lista de documentos a partir de la interfaz de la aplicación es necesario el siguiente código:

```
Documents* Docs;
HRESULT hr = App->get_Documents(&Docs);
if (SUCCEEDED(hr))
{
    // Trabajar con la lista de documentos
    Docs->Release();
}
```

En el ejemplo anterior incluso he evitado introducir el control de excepciones, y ya vemos que para recuperar una simple propiedad necesitamos un montón de líneas de código. Afortunadamente, cuando C++ Builder importa una biblioteca de tipos también define interfaces inteligentes que automatizan el uso de *AddRef* y *Release*, y

define adicionalmente propiedades en las mismas que corresponden a las definiciones de propiedades del lenguaje IDL. El siguiente método muestra cómo utilizar las interfaces inteligentes:

```
void __fastcall TForm1::Button3Click(TObject *Sender)
{
    TCOM_Application App = CoApplication_::Create();
    App.Visible = true;
    TCOM_Document Doc = TCOMDocuments(App.Documents_).Add();
    Doc.Activate();
    TCOMRange R = Doc.Content;
    WideString texto = "Mira lo que hago con Word\n";
    R.InsertAfter(texto);
}
```

Las nuevas clases, como *TCOM_Document*, están basadas en plantillas como *TCOM-Interface* definidas por la ATL.

Existe una técnica intermedia entre el uso de variantes y el de interfaces duales. Consiste en evitar las llamadas a *GetIDsOfNames*, al conocer directamente los identificadores *dispatch* de los métodos soportados. Cuando se importa una biblioteca de tipos, C++ Builder crea también clases para este tipo de llamadas. Sin embargo, en la mayoría de los casos nos interesa utilizar directamente la interfaz dual y no pasar por una técnica que no es ni la más rápida ni la más sencilla.

Eventos

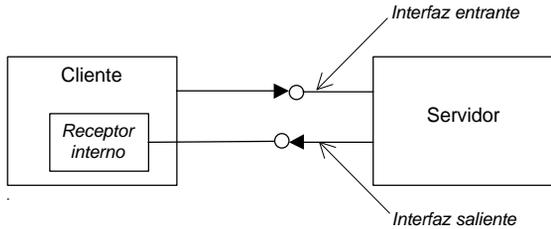
El tipo de comunicación más frecuente en la automatización OLE es unidireccional: el cliente da órdenes al objeto servidor. El paso siguiente para facilitar la programación es que el servidor pueda enviar eventos al cliente. Suponga que, en vez de enfrentarse a una máquina de bebidas, tiene usted que plantar cara a la máquina expendedora de pizzas. La máquina de bebidas podía soportar una interfaz simple con este método, declarado en IDL:

```
HRESULT PedirBebida([in] BSTR producto, [out] IBebida bebida);
```

El tiempo que transcurre entre que pulsemos el botón y que la máquina escupa el bote correspondiente es despreciable. *PedirBebida* puede hacernos esperar hasta entonces. Pero la máquina de pizzas tiene un retardo inherente: hay que seleccionar los ingredientes, mezclarlos y ponerlos en el horno. Hasta que la pizza esté en su punto, ¿qué hacemos, silbar con las manos en los bolsillos aparentando indiferencia? Mejor que eso, podemos definir un método de petición más sencillo aún:

```
HRESULT PedirPizza([in] IIngredientes bazofia);
```

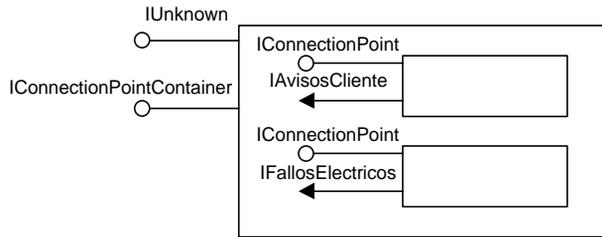
Cuando la pizza esté lista, queremos que la máquina nos avise asincrónicamente para entonces actuar. Si el objeto expendedor de pizzas fuera un componente VCL, implementaríamos el aviso mediante un evento. ¿Cómo envían eventos los componentes de la VCL? Utilizando punteros a métodos. ¿Existen punteros a métodos en COM? No directamente ... pero sí agrupados dentro de interfaces. Entonces, para que un servidor pueda enviar métodos a un cliente, debe llamar a métodos de una interfaz implementada por el cliente, como se muestra en el siguiente diagrama.



El servidor es el encargado de definir la interfaz de envío de eventos, conocida con el nombre de *interfaz saliente (outgoing interface)*, y puede tener todos los métodos que se le ocurra al programador. Eso sí, el que inventa la interfaz de salida tiene que tener en cuenta que el cliente no puede escribir directamente en la memoria del servidor, como sí puede suceder en la interacción con un componente VCL. Tiene entonces que proporcionar algún mecanismo para que el cliente se suscriba a los envíos de determinada interfaz de eventos.

El mecanismo de suscripción puede ser todo lo arbitrario según lo decida el implementador del servidor. Pero es de agradecer la existencia de una técnica de conexión estándar definida por COM, como son los *puntos de conexión*, representados mediante las interfaces *IConnectionPoint* e *IConnectionPointContainer*. En el capítulo siguiente tendremos la ocasión de ver cómo C++ Builder ayuda a la implementación de estas interfaces en los servidores de automatización. Aquí explicaremos cómo utilizarlas desde una aplicación cliente.

En pocas palabras, si un servidor desea soportar una o más interfaces de eventos debe implementar una o más interfaces *IConnectionPoint*, que servirán para que los clientes se suscriban a las mismas. Sin embargo, estas interfaces de suscripción no se exportan directamente: el cliente no debe poder obtenerlas mediante llamadas al método *QueryInterface*. La encargada de publicarlas es la interfaz *IConnectionPoint*, que ofrece métodos para localizar un punto de conexión dado su identificador único de interfaz:



Cuando, más adelante, muestre un ejemplo de cómo recibir eventos de Word 97, aprovecharé la rutina *InterfaceConnect* de la VCL para enlazar al emisor y al receptor de una interfaz de eventos. Es muy probable que la ATL ofrezca una función de este tipo, pero ni siquiera me he tomado la molestia de buscarla. Sin embargo, la mejor forma de mostrar cómo se realiza la conexión al nivel más detallado, es conveniente estudiar cómo se implementaría un sustituto de *InterfaceConnect* en el propio C++ Builder:

```
void IntfConn(IUnknown* Emisor, REFIID riid,
             IUnknown* Receptor, unsigned long& Cookie)
{
    IConnectionPointContainer *CPC;

    Cookie = 0;
    HRESULT hr = Emisor->QueryInterface(
        IID_IConnectionPointContainer, (void**) &CPC);
    if (SUCCEEDED(hr))
    {
        IConnectionPoint *CP;
        hr = CPC->FindConnectionPoint(riid, &CP);
        if (SUCCEEDED(hr))
        {
            CP->Advise(Receptor, &Cookie);
            CP->Release();
        }
        CPC->Release();
    }
}
```

Primero se intenta buscar la interfaz genérica *IConnectionPointContainer* dentro del objeto emisor. Si el intento triunfa, se le pide a la lista de puntos de conexión encontrada que busque un punto de conexión para la interfaz de eventos identificada mediante *riid*. Si existe tal punto de conexión, se le “avisa” mediante el método *Advise* para que envíe notificaciones al nuevo receptor. *Advise* devuelve un identificador para la conexión, que hemos almacenado en el parámetro *Cookie*.

La función *InterfaceDisconnect* de la VCL invierte la acción de *InterfaceConnect*. Su equivalente en C++ sería como sigue:

```
void IntfDisconn(IUnknown* Emisor, REFIID riid,
                unsigned long& Cookie)
```

```

{
    if (Cookie != 0)
    {
        IConnectionPointContainer *CPC;
        HRESULT hr = Emisor->QueryInterface(
            IID_IConnectionPointContainer, (void**) &CPC);
        if (SUCCEEDED(hr))
        {
            IConnectionPoint *CP;
            hr = CPC->FindConnectionPoint(riid, &CP);
            if (SUCCEEDED(hr))
            {
                hr = CP->Unadvise(&Cookie);
                if (SUCCEEDED(hr)) Cookie = 0;
                CP->Release();
            }
            CPC->Release();
        }
    }
}

```

Los parámetros de *IntfDisconn*, que imitan a los de *InterfaceDisconnect*, son casi iguales a los de la rutina de conexión, exceptuando el puntero al receptor, que ahora es innecesario. Incluso es similar el prólogo de las rutinas, y lo que cambia es que para interrumpir la conexión se llama a *Unadvise*, con el identificador de conexión generado anteriormente.

Una importante propiedad de *Advise* es que, si el implementador así lo desea, puede permitir que varios clientes se abonen simultáneamente a la misma interfaz de eventos. Un evento de la VCL solamente puede contener un puntero a método, por lo que puede soportar sólo un cliente a la vez. Pero un servidor COM puede mantener una lista de clientes suscritos, o decidir mantener un solo cliente simultáneo, denegando las restantes conexiones.

Escuchando a Word

Como ejercicio, haremos que una aplicación escrita en C++ Builder reciba notificaciones enviadas por Word97. La clase *Word.Application* soporta la interfaz de salida *ApplicationEvents*, cuyo evento más importante es *DocumentChange*, y se dispara cada vez que cambia el documento activo de la aplicación. Para recibir estos eventos con mayor facilidad necesitaremos definir un pequeño objeto auxiliar que actúe como receptor. Esta es la declaración de la clase receptora:

```

class TReceptor : public IDispatch
{
private:
    int Cookie;
    IUnknown *Source;
public:
    TReceptor(IUnknown *ASource);

```

```

~TReceptor();
STDMETHODIMP_(ULONG) AddRef();
STDMETHODIMP_(ULONG) Release();
STDMETHODIMP QueryInterface(REFIID, LPVOID*);
STDMETHODIMP GetTypeInfoCount(UINT *pctinfo);
STDMETHODIMP GetTypeInfo(UINT iTInfo, LCID lcid,
    ITypeInfo **ppTInfo);
STDMETHODIMP GetIDsOfNames(REFIID riid, LPOLESTR *rgszNames,
    UINT cNames, LCID lcid, DISPID *rgDispId);
STDMETHODIMP Invoke(DISPID dispIdMember, REFIID riid, LCID lcid,
    WORD wFlags, DISPPARAMS *pDispParams, VARIANT *pVarResult,
    EXCEPINFO *pExcepInfo, UINT *puArgErr);
};

```

La clase *TReceptor* es utilizada un poco más adelante por el formulario principal de la aplicación:

```

class TForm1 : public TForm
{
__published: // IDE-managed Components
    TListBox *ListBox1;
private: // User declarations
protected:
    TCOM_Application WordApp;
    TReceptor *receptor;
public: // User declarations
    __fastcall TForm1(TComponent* Owner);
    __fastcall ~TForm1();
    void __fastcall SignalEvent(const AnsiString s);
};

```

Durante la construcción y la destrucción del receptor activamos o rompemos el enlace con el emisor:

```

TReceptor::TReceptor(IUnknown *ASource) : Cookie(0), Source(ASource)
{
    InterfaceConnect(Source, DIID_ApplicationEvents,
        (IUnknown*) this, Cookie);
}

TReceptor::~TReceptor()
{
    InterfaceDisconnect(Source, DIID_ApplicationEvents, Cookie);
}

```

La implementación de *QueryInterface* es muy simple:

```

STDMETHODIMP TReceptor::QueryInterface(REFIID riid, LPVOID* ppv)
{
    if (riid == IID_IUnknown ||
        riid == IID_IDispatch ||
        riid == DIID_ApplicationEvents)
        *ppv = (LPVOID) this;
    else
        return E_NOINTERFACE;
}

```

```

    return S_OK;
}

```

El núcleo de la clase es la implementación de *Invoke*. Aquí debemos consultar el fichero importado desde la propia biblioteca de tipos de Word para comprobar cuáles son los identificadores numéricos asociados a los eventos que nos interesan. Veremos, por ejemplo, que la interfaz *ApplicationEvents* recibe la notificación 3 para indicar que ha cambiado el documento activo:

```

STDMETHODIMP TReceptor::Invoke(DISPID dispIdMember, REFIID riid,
    LCID lcid, WORD wFlags, DISPPARAMS *pDispParams,
    VARIANT *pVarResult, EXCEPINFO *pExcepInfo, UINT *puArgErr)
{
    if (dispIdMember == 3)
        Form1->SignalEvent("Ha cambiado el documento activo");
    return S_OK;
}

```

A los métodos restantes, heredados de *IUnknown* y de *IDispatch*, les podemos asignar una implementación mínima:

```

STDMETHODIMP_(ULONG) TReceptor::AddRef()
{
    return 2;
}

STDMETHODIMP_(ULONG) TReceptor::Release()
{
    return 1;
}

STDMETHODIMP TReceptor::GetTypeInfoCount(UINT *pctinfo)
{
    *pctinfo = 0;
    return S_OK;
}

STDMETHODIMP TReceptor::GetTypeInfo(UINT iTInfo, LCID lcid,
    ITypeInfo **ppTInfo)
{
    return E_NOTIMPL;
}

STDMETHODIMP TReceptor::GetIDsOfNames(REFIID riid,
    LPOLESTR *rgszNames, UINT cNames, LCID lcid, DISPID *rgDispId)
{
    return E_NOTIMPL;
}

```

Y ya estamos listos para aprovechar la nueva clase auxiliar en nuestra aplicación, como podemos ver en el constructor y en el destructor de la ventana principal:

```

__fastcall TForm1::TForm1(TComponent* Owner) : TForm(Owner),
    WordApp(CoApplication_::Create())

```

```

{
    receptor = new TReceptor(WordApp.Application__);
    WordApp.Visible = true;
}

__fastcall TForm1::~TForm1()
{
    delete receptor;
}

```

Para que el usuario se entere del disparo del evento, he utilizado un sencillo cuadro de listas:

```

void TForm1::SignalEvent(const AnsiString s)
{
    ListBox1->Items->Add(s + " -> " + TimeToStr(Now()));
}

```

Para poder sacar un poco más de partido de los eventos de Word, sería también necesario interceptar la interfaz de eventos *DocumentEvents*, del objeto *Document*. La dificultad consiste en atender al cambio de documento activo para establecer y romper la conexión con el receptor. Como a estas alturas usted se ha convertido en todo en un experto en la materia, esta tarea se la dejo como desayuno.

Automatización OLE: servidores

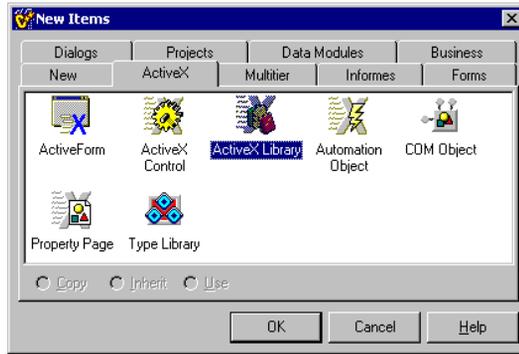
AHORA ESTAMOS EN CONDICIONES DE explicar cómo se crean servidores de automatización con C++ Builder. Al ser bastante sencilla la programación de este tipo de aplicaciones gracias a la existencia de asistentes, el énfasis de la presentación se desplazará de la teoría a la práctica. Incluiremos usos prácticos tanto de los servidores como de sus clientes. Aunque no hay muchas diferencias en la programación COM entre las versiones 3 y 4 de C++ Builder, los ejemplos estarán basados principalmente en C++ Builder 4, como es de suponer.

Informes automatizados

Un servidor de automatización debe ofrecer clases que implementen la interfaz *IDispatch*. Utilizaremos la siguiente excusa para su creación: tenemos una aplicación de bases de datos y hemos definido una serie de informes con QuickReport sobre las tablas con las que trabaja el programa. Pero prevemos que el usuario querrá más adelante nuevos tipos de informes. ¿Qué hacemos cuando el usuario tenga una idea genial de ese tipo? ¿Abrimos la aplicación en canal para enlazar el nuevo informe? Claro que no. Una solución es definir los informes dentro de DLLs, y permitir la carga dinámica de las mismas. La otra, que exploraremos aquí, es programar los nuevos informes como servidores de automatización, y que la aplicación pueda imprimirlos llamando a métodos exportados tales como *Print*, *Preview* y *SetupPrint*.

QuickReport será estudiado unos cuantos capítulos más adelante. Pero no se preocupe ahora por ello, pues utilizaremos asistentes para generar los informes de prueba del servidor.

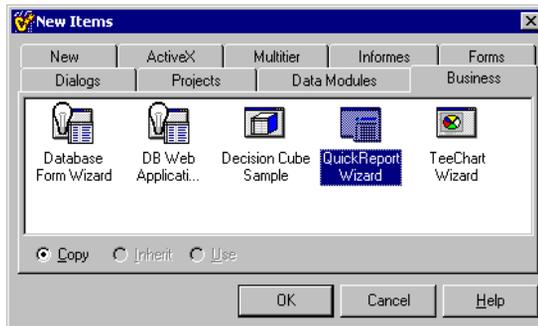
Comencemos por la creación de un servidor de este tipo. La primera decisión consiste en elegir entre un servidor dentro del proceso (una DLL) o fuera del proceso (un EXE). Por supuesto, una DLL es más eficiente, por lo que vamos al Depósito de Objetos, seleccionamos la página *ActiveX* y pulsamos el icono *ActiveX Library*.



Esta operación genera un esqueleto de DLL, que guardamos con el nombre *Informes*. En el capítulo sobre servidores COM vimos que estas DLLs implementan y exportan cuatro funciones:

- *DllRegisterServer*: registra el servidor.
- *DllUnregisterServer*: elimina las entradas del registro.
- *DllGetClassObject*: paso inicial del proceso de construcción de instancias.
- *DllCanUnloadNow*: controla el tiempo de vida del módulo.

También necesitaremos un informe. Para los propósitos de este ejercicio, da lo mismo qué informe utilicemos y cómo lo generemos. Utilizaremos el *QuickReport Wizard*, de la página *Business* del Depósito de Objetos:



El asistente nos permite seleccionar una tabla de cualquiera de los alias registrados por el BDE, y elegir qué campos queremos imprimir. Utilice cualquier alias, preferentemente uno correspondiente a una base de datos de escritorio, y cualquier tabla de la misma. Yo he elegido imprimir todos los campos de la tabla *customer* del alias predefinido *bcbdemo*. Al finalizar el proceso, nos encontramos una nueva unidad, que guardaremos con el nombre de *ClientesRpt*, y un formulario *Form1* que contiene una serie de componentes de impresión. El formulario, en sí, nunca se mostrará directamente en pantalla, sino que se limitará a servir de contenedor.

Como el formulario reside dentro de una DLL, no se crea automáticamente al cargarse el módulo en memoria. Para simplificar la creación y destrucción automática de este formulario declararemos una serie de métodos estáticos en la clase *TForm1*:

```
class TForm1 : public TForm
{
    __published: // IDE-managed Components
        // ...
    private: // User declarations
    public: // User declarations
        __fastcall TForm1(TComponent* Owner);
        static void Print();
        static void Preview();
        static void SetupPrint();
};
```

Print imprime directamente el informe. *Preview* muestra una ventana modal con la vista preliminar del informe. Dentro de esta vista hay un botón para imprimir el informe. Por último, *SetupPrint* muestra primero un diálogo de impresión antes de proceder a la impresión. La implementación de los tres métodos es muy sencilla:

```
void TForm1::Print()
{
    std::auto_ptr<TForm1> f(new TForm1(0));
    f->QuickRepl->Print();
}

void TForm1::Preview()
{
    std::auto_ptr<TForm1> f(new TForm1(0));
    f->QuickRepl->PreviewModal();
}

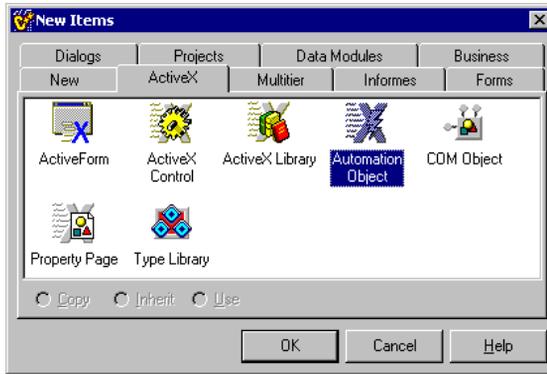
void TForm1::SetupPrint()
{
    std::auto_ptr<TForm1> f(new TForm1(0));
    f->QuickRepl->PrinterSetup();
    if (f->QuickRepl->Tag == 0)
        f->QuickRepl->Print();
}
```

Aquí *QuickRepl1* apunta al componente principal del informe, que es el encargado de la impresión y de la vista preliminar. Estoy realizando un truco sucio en *SetupPrint* que explicaré en el capítulo dedicado a QuickReport.

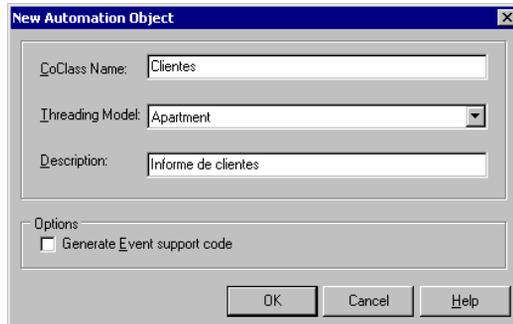
El objeto de automatización

Hasta el momento, hemos logrado una DLL algo especial que contiene una clase con un informe. Para que esta DLL se convierta realmente en un servidor de automatización y que cualquier programa pueda aprovechar el informe que hemos definido, debemos ejecutar el Depósito de Objetos, seleccionar y aceptar el icono *Automation*

Object, de la página *ActiveX*. De esta manera crearemos una clase de componentes que implementará la clase *IDispatch*, para que pueda aceptar macros de controladores de automatización.

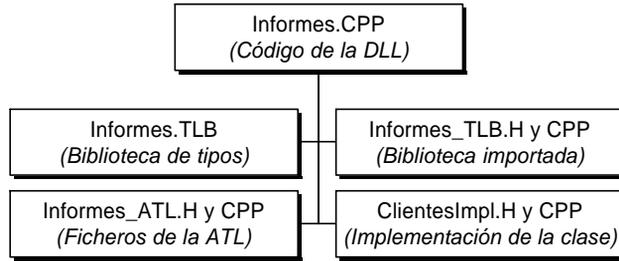


El diálogo que aparece al ejecutar el asistente nos ayuda a configurar las principales características de la clase que vamos a implementar. La primera de ellas es el nombre de la clase, donde tecleamos *Clientes*. Esto implica que el identificador de programa de la clase será *Informes.Clientes*, esto es, la concatenación del nombre del servidor con el nombre de la clase.



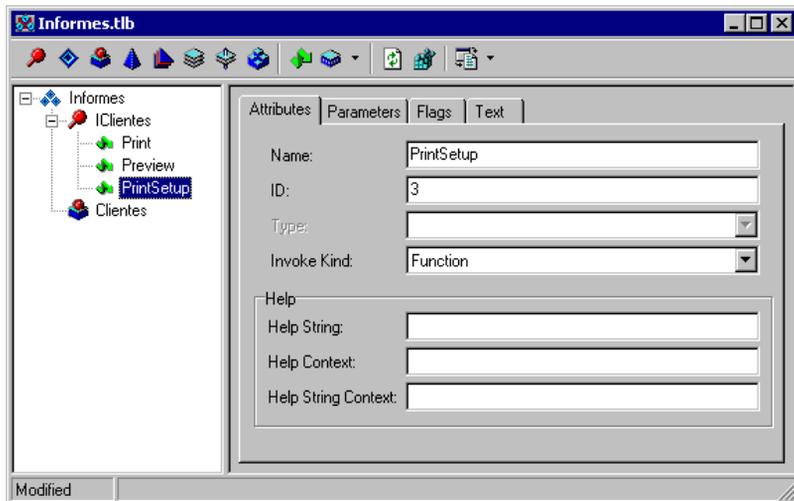
A continuación, se nos pide el modelo de concurrencia. Un poco más adelante explicaremos en qué consiste, pero ahora utilizaremos el modelo *Apartment*. También hace falta una descripción para la clase, y finalmente se nos pregunta si queremos generar código para soportar eventos. Si activáramos esta opción, nuestra clase implementaría la ya conocida interfaz *IConnectionPointContainer*, pero para los propósitos de este ejemplo no nos interesa enviar eventos.

Cuando aceptamos el diálogo de creación de objetos de automatización, C++ Builder genera automáticamente varios ficheros y los incluye dentro del proyecto activo, para que hagan compañía a *ClientesRpt*, que contiene el formulario del informe:



En primer lugar, se crea el fichero *Informes_ATL.cpp* y su cabecera correspondiente. Ahí se define e implementa la variable global *_Module*, de tipo *CCOMModule*, y se incluyen dentro del proyecto las declaraciones de la ATL. Así ya hemos garantizado los pasos mecánicos de grabación en el registro y exposición de la fábrica de clases.

Se incluye además una biblioteca de tipos para la aplicación, que es una entidad global al proyecto. Como nuestro proyecto se llama *Informes*, la biblioteca se guarda en el fichero *Informes.tlb*. Como es de suponer, también se crea el fichero de cabecera correspondiente a la biblioteca, con el nombre *Informes_TLB.h*. Además, se genera un fichero *Informes_TLB.cpp* para la que contenga la inicialización de variables globales con los identificadores de clases, interfaces y bibliotecas definidos. No debemos editar directamente ninguno de estos dos ficheros, sino a través del editor visual de las bibliotecas de tipos, que puede activarse explícitamente mediante el comando de menú *View | Type Library*:



Seleccionamos en el árbol de la izquierda del editor de la biblioteca de tipos el nodo correspondiente a la interfaz *ICientes*. Mediante el menú de contexto, o mediante los iconos de la barra de herramientas, añadimos tres métodos, a los cuales nombraremos *Print*, *Preview* y *SetupPrint*. Ninguno de los tres métodos tiene parámetros, ni valo-

res de retorno; en caso contrario, deberíamos especificarlos en la segunda página de las propiedades de estos nodos (*Parameters*).

Sólo nos queda proporcionar una implementación a esta interfaz. Y aquí C++ Builder ha vuelto a echarnos una mano, pues ha generado una unidad, *CientesImpl*, con la declaración de la clase *TCientesImpl*:

```

class ATL_NO_VTABLE TCientesImpl :
public CComObjectRootEx<CComSingleThreadModel>,
public CComCoClass<TCientesImpl, &CLSID_Cientes>,
public IDispatchImpl<ICientes, &IID_ICientes, &LIBID_Informes>
{
public:
    TCientesImpl()
    {
    }

    // Data used when registering Object
    DECLARE_THREADING_MODEL(otApartment);
    DECLARE_PROGID("Informes.Cientes");
    DECLARE_DESCRIPTION("Informe de clientes");

    // Function invoked to (un)register object
    static HRESULT WINAPI UpdateRegistry(BOOL bRegister)
    {
        TTypedComServerRegistrarT<TCientesImpl>
        regObj(GetObjectCLSID(), GetProgID(), GetDescription());
        return regObj.UpdateRegistry(bRegister);
    }

    BEGIN_COM_MAP(TCientesImpl)
        COM_INTERFACE_ENTRY(ICientes)
        COM_INTERFACE_ENTRY(IDispatch)
    END_COM_MAP()

    // ICientes
public:
    STDMETHOD(Preview());
    STDMETHOD(Print());
    STDMETHOD(PrintSetup());
};

```

La implementación de los tres métodos es elemental, pues nos limitaremos a pasarle la patata caliente a la clase *TForm1*:

```

STDMETHODIMP TCientesImpl::Preview()
{
    TForm1::PreviewModal();
    return S_OK;
}

STDMETHODIMP TCientesImpl::Print()
{
    TForm1::Print();
    return S_OK;
}

```

```

STDMETHODIMP TClientesImpl::PrintSetup()
{
    TForm1::SetupPrint();
    return S_OK;
}

```

La parte cliente

La creación de un controlador de automatización que utilice al servidor definido anteriormente es sumamente sencilla. Creamos una nueva aplicación, con un solo formulario. Colocamos un cuadro de edición sobre el mismo, para que el usuario pueda teclear el nombre de la clase de automatización, un botón para mostrar la vista preliminar del informe y otro para configurar la impresora e imprimir. Recuerde que el objetivo del ejemplo anterior era la creación de informes genéricos, que podían ejecutarse desde una aplicación con sólo conocer el nombre de la clase. Por supuesto, en una aplicación real emplearíamos un mecanismo más sofisticado para ejecutar los informes genéricos, que quizás contara con la creación de claves especiales en el Registro durante la instalación de los informes. Pero aquí nos basta con una sencilla demostración de la técnica.



La respuesta al evento *OnClick* de los botones que hemos situado en el formulario es la siguiente:

```

void __fastcall TForm1::Button1Click(TObject *Sender)
{
    Variant Rpt = Variant::CreateObject(Edit1->Text);
    Rpt.Exec(Procedure("Preview"));
}

void __fastcall TForm1::Button1Click(TObject *Sender)
{
    Variant Rpt = Variant::CreateObject(Edit1->Text);
    Rpt.Exec(Procedure("SetupPrint"));
}

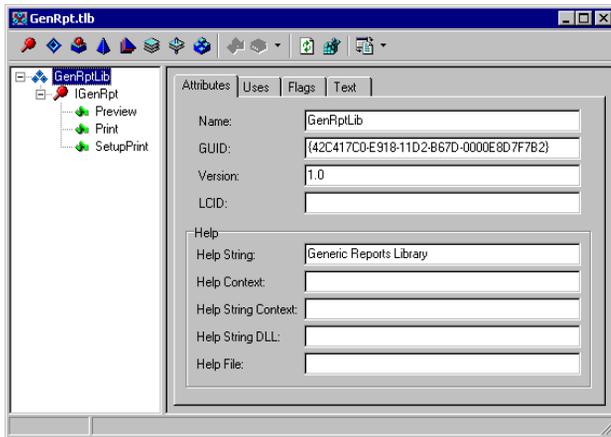
```

Declarando una interfaz común

Pero, un momento ... ¿no habíamos quedado en que íbamos a crear un mecanismo de impresión genérico? Sí, y el ejemplo que hemos desarrollado permite la impresión genérica. Usted solamente tiene que indicar el nombre de una clase que implemente los métodos *Print*, *Preview* y *SetupPrint*, y la aplicación cliente se encarga de crear un objeto de dicha clase y aplicarle el método correspondiente. ¿Trampa? Claro que la hay: el polimorfismo se logra mediante llamadas a macros. Si hubiéramos deseado aprovechar la interfaz dual *IClientes* se habría roto la magia polimórfica pues, si utilizamos el asistente de creación de objetos de automatización, cada servidor de impresión implementará una clase diferente con una interfaz diferente, aunque cada interfaz exporte los mismos tres métodos.

¿Es esto inevitable? Por supuesto que no. Podemos definir una interfaz genérica, a la que llamaremos *IGenRpt*, que exporte los tres métodos de impresión, y podemos hacer que cada servidor de informes cree clases diferentes, eso sí, pero que implementen la misma interfaz.

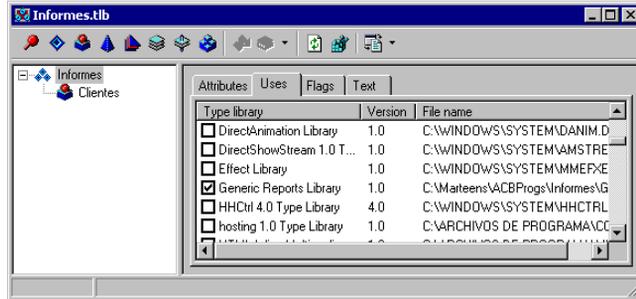
Para crear la interfaz común podemos seguir varias alternativas: crear un fichero IDL y compilarlo, crear la interfaz mediante el editor de bibliotecas de tipos de C++ Builder, etc. Aquí vamos a seguir la última vía mencionada. Primero cerramos todos los ficheros abiertos en el Entorno de Desarrollo (*File | Close all*). Luego activamos el Depósito de Objetos (*File | New*), y en la página *ActiveX* seleccionamos *Type Library*:



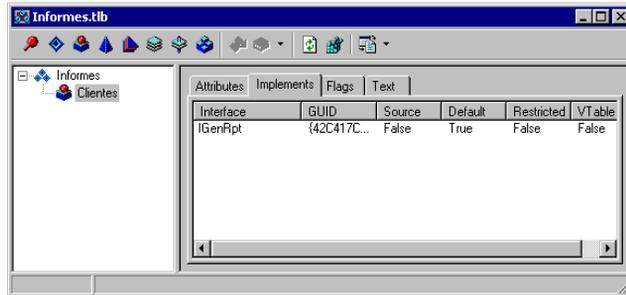
Aparecerá el conocido editor gráfico de bibliotecas de tipos, y en él crearemos la nueva interfaz *IGenRpt*, definiendo para ella los métodos *Print*, *Preview* y *SetupPrint*. Cambie el nombre de la biblioteca en el nodo raíz del panel de la izquierda a *GenRptLib*, y modifique la cadena de ayuda asociada. Cuando haya terminado, guarde el fichero, lo cual servirá también para crear los ficheros *GenRptLib_TLB* con extensiones *cpp* y *h*. Finalmente, registre la biblioteca pulsando el botón correspondiente

de la barra de herramientas del editor. Este paso es crucial para que C++ Builder pueda hacer referencias a la nueva biblioteca desde otro proyecto.

Para modificar el servidor de informes existente son necesarios dos pasos. Primero tenemos que incluir la nueva biblioteca de tipos en la biblioteca del servidor. Seleccionamos el nodo *Informes* y activamos la página *Uses* del editor. Pulsamos el botón derecho del ratón y ejecutamos el comando del menú local *Show All Type Libraries*, para que se muestren todas las bibliotecas registradas. Cuando localice la biblioteca *GenRptLib* utilizando la cadena de ayuda, selecciónela:



A continuación hay que eliminar la interfaz *IClientes* de la lista de interfaces implementadas por la clase *Clientes*, lo cual se realiza en la página *Implements* del nodo *Clientes*. Para terminar, se quita también la definición de *IClientes* del árbol de la izquierda, se añade *IGenRpt* a la lista de interfaces implementadas por la clase *Clientes* y se marca como la interfaz por omisión (*Default*):



Añada ahora al proyecto el fichero *GenRptLib_TLB.cpp*, pues ahí es donde se inicializa la variable que contiene el identificador de la interfaz genérica *IGenRpt*. Y ya estamos en condiciones de guardar el proyecto y volverlo a compilar. Posiblemente sea necesario retocar algo el código fuente generado para C++, sobre todo algunas referencias a la extinta interfaz *IClientes* que seguirán vagando como almas en pena por todo nuestro código.

¿Cómo queda ahora la aplicación que utiliza los objetos de impresión? En primer lugar, también hay que incluir el fichero de importación de la biblioteca de tipos *GenRptLib*, e incluir la cabecera dentro de la unidad del formulario principal. Para la vista preliminar del informe hace falta el siguiente código:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    GUID clsid = ProgIDToClassID(Edit1->Text);
    IGenRpt *rpt;
    OLECHECK(CoCreateInstance(clsid, NULL, CLSCTX_ALL, IID_IGenRpt,
        (void**) &rpt));
    rpt->Preview();
    rpt->Release();
}
```

Observe que la activación del informe es completamente genérica. Cada servidor implementará una clase distinta, de la cual tendremos que saber solamente su identificador de programa, no el de clase. La traducción la estamos realizando mediante *ProgIDToClassID*. A continuación, se crea una instancia de esa clase, y se pide directamente el puntero a su interfaz *IGenRpt*. Como pueden existir errores, se verifica la operación mediante la macro *OLECHECK*. Si la obtención de la interfaz triunfa, se ejecuta el método *Preview* sobre la misma, y finalmente se libera el objeto por medio del método *Release*.

Quizás hubiera sido más sencillo respetar la interfaz original *IClientes*, y hacer que la clase *Clientes* implementara “además” la interfaz *IGenRpt*. Pero el generador de código de C++ Builder enloqueció cuando le propuse esta posibilidad, que es en realidad bastante inocua.

Modelos de instanciación

¿Qué tal un poco de teoría, para variar? Comencemos por explicar qué es un *modelo de instanciación*. Tomemos como punto de partida un servidor dentro del proceso. Cuando una aplicación necesita una clase implementada dentro del servidor, se carga la DLL dentro del espacio de memoria de la aplicación cliente para crear el objeto. Si otro cliente necesita simultáneamente un objeto de la misma clase, el proceso de carga se repite en un espacio de direcciones distinto.

¿Qué pasa cuando el servidor es un fichero ejecutable? Cuando el cliente solicita un objeto de la clase COM, el sistema operativo busca dentro del Registro y encuentra el programa servidor asociado con la clase y lo lanza. El ejecutable, por su parte, debe llamar a la función *CoRegisterClassObject* para que COM tenga a su alcance las fábricas de clase implementadas. Luego, el programa entra dentro de su ciclo de mensajes.

Ya podemos imaginar lo que sucede cuando se pide la primera instancia de la clase: COM busca en la lista de fábricas de clase, selecciona la adecuada y ejecuta sobre ella el método *CreateInstance*. ¿Qué sucede si ese mismo cliente, o cualquier otra aplicación, solicita una instancia adicional? Depende de cómo se haya registrado la clase. Veamos primero el prototipo de la función de registro de clases:

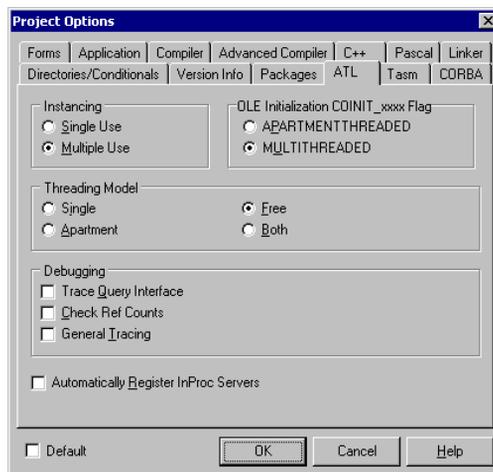
```
STDAPI CoRegisterClassObject(REFCLSID rclsid, IUnknown *pUnk,
    DWORD dwClsContext, DWORD flags, LPDWORD *lpdwRegister);
```

El parámetro que nos interesa ahora es el cuarto parámetro, en el cual se puede pasar uno de los siguiente valores enumerativos:

```
typedef enum tagREGCLS {
    REGCLS_SINGLEUSE      = 0,
    REGCLS_MULTIPLEUSE   = 1,
    REGCLS_MULTISEPARATE = 2,
} REGCLS;
```

Si utilizamos *REGCLS_SINGLEUSE*, una vez que se crea el primer objeto, COM “esconde” la fábrica de clases, para que no puedan crearse más instancias de la clase. De este modo, solamente se permite globalmente un solo objeto de la clase. Lo normal es, sin embargo, que se utilice *REGCLS_MULTIPLEUSE*. En tal caso, la misma instancia del programa puede crear otras instancias de la clase; eso sí, cada una en su propio hilo paralelo. Todos estos objetos pueden compartir el espacio de direcciones global de la aplicación. Por último, *REGCLS_MULTISEPARATE* fuerza a COM a que ejecute otra vez la aplicación para obtener otro objeto: un objeto equivale a una instancia de la aplicación.

¿Cómo se le indica a la ATL el modelo de instanciación que necesitamos? Cuando el proyecto activo contiene objetos COM, se activa la página *ATL* dentro del diálogo de opciones del proyecto (*Project | Options*):



También se puede recurrir a la “fuerza bruta”. Por omisión, se utiliza *multiple use*, pero si necesitamos *multi separate* hay que definir el siguiente símbolo:

```
#define _ATL_SINGLEUSE_INSTANCING
```

El mejor sitio para la definición es al principio del código fuente del proyecto, que es el primer fichero que se compilará dentro de la aplicación. Recuerde que el modelo de instanciación solamente es importante para servidores ejecutables, pero no para servidores dentro del proceso.

Modelos de concurrencia

Los modelos de concurrencia aparecieron con DCOM, en relación con la posibilidad de activar un servidor situado en otro puesto de la red. Por este motivo, los libros clásicos sobre COM de la primera generación no hacen referencia al problema. Para colmo de males, la explicación que incluyen hasta el momento los manuales de C++ Builder y Delphi es confusa, cuando menos. Dentro de mis humildes posibilidades, intentaré organizar y condensar parte de la información que se encuentra dispersa sobre la concurrencia en el Modelo de Objetos Componentes.

Probablemente usted tenga aunque sea unas nociones mínimas acerca de la estructura de procesos de Windows. Sabrá entonces que en Windows pueden existir simultáneamente varios procesos concurrentes, y que cada proceso es el propietario de uno o más hilos (*threads*) de ejecución. Pues ahora le presento un nuevo personaje: los *apartamentos* (*apartments*). Un apartamento no es un proceso, ni un hilo; tampoco es un pájaro, un avión o Superman. Un hilo de una aplicación COM debe ejecutarse exactamente en un apartamento, pero un apartamento puede ser utilizado simultáneamente por varios hilos. La existencia de un apartamento impone restricciones a los objetos que habitan en su interior, relacionadas con la forma en que pueden comunicarse con otros apartamentos. Microsoft denominó originalmente a los apartamentos con el nombre de *contexto de ejecución*; en mi opinión, este nombre es preferible, ya que no posee connotaciones inmobiliarias...

Más información sobre los apartamentos. Un proceso puede contener dos tipos de apartamentos: los MTA (*multi-threaded apartment*) y los STA (*single-threaded apartment*). Solamente se permite un MTA por proceso, pero se admite la presencia de varios STA. Como sus nombres sugieren, dentro de un MTA pueden ejecutarse varios hilos, pero no dentro de un STA.

Para que un hilo pueda utilizar COM debe llamar a una de las siguientes funciones:

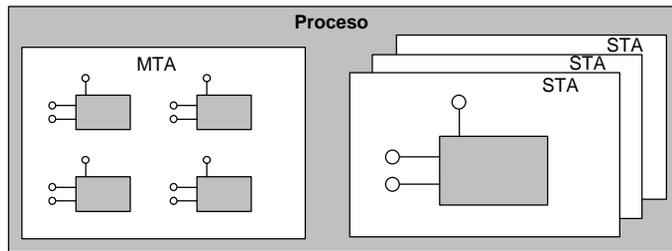
```
HRESULT CoInitialize(LPVOID pvReserved);
HRESULT OleInitialize(LPVOID pvReserved);
HRESULT CoInitializeEx(LPVOID pvReserved, DWORD dwCoInit);
```

El primer parámetro de las tres funciones anteriores siempre debe ser 0. Las dos primeras indican que el hilo activo se ejecutará dentro de un nuevo STA. La tercera función es un añadido posterior de Microsoft, y controla el tipo de apartamento de acuerdo al valor pasado en el segundo parámetro:

<i>dwCoInit</i>	Tipo de apartamento
<i>COINIT_APARTMENTTHREADED</i>	STA
<i>COINIT_MULTITHREADED</i>	MTA

¡Eh, cuidado! Hace un par de capítulos programamos un servidor COM sencillo, y no llamamos a *CoInitialize* para nada! Es verdad, pero recuerde que se trataba de un servidor dentro del proceso, es decir, una DLL, por lo que la llamada en cuestión la realizaba el cliente.

¿Para qué se han introducido los apartamentos? ¿Sólo para complicarnos la vida? La respuesta es que cuando un objeto COM reside en un STA, la activación de sus métodos se realiza indirectamente a través de una cola de mensajes propia del apartamento. Si varios hilos clientes intentan acceder a los métodos del objeto no se produce conflicto alguno, pues la cola de mensajes serializará automáticamente las llamadas. Por lo tanto, los métodos de la clase no tendrán que utilizar primitivas de sincronización (secciones críticas, por ejemplo) para acceder a los datos del objeto. Eso sí, si el servidor es un ejecutable que puede suministrar varios objetos dentro de la misma instancia, no hay una protección similar para los datos globales a todos los objetos.



Los servidores locales y remotos pueden indicar el modelo de concurrencia de sus objetos controlando las llamadas a *CoInitializeEx*. Pero si el servidor es una DLL, es el cliente el único que puede llamar a la función mencionada. ¿Qué sucede si el servidor supone que va a ejecutarse en determinadas condiciones de concurrencia, y resulta que el cliente utiliza otras?

Por este motivo, los servidores dentro del proceso deben indicar su modelo de concurrencia en el Registro de Configuraciones, en la subclave *Threading Model*. Entonces el sistema operativo puede tomar las precauciones necesarias al cargar el servidor

para evitar conflictos entre los modelos de concurrencia del cliente y del servidor. Una de estas medidas puede incluso ser la delegación de las llamadas del cliente a un *proxy*, aunque el objeto servidor resida en el mismo espacio de memoria de la aplicación que lo utiliza.

La función *CoInitializeEx* es una extensión añadida para DCOM, y como todos sabemos Windows 95 no soporta objetos distribuidos ... a no ser que instalemos un parche suministrado por Microsoft. Si usted no instala este parche, no podrá ejecutar (ni siquiera registrar) los ejemplos de ActiveX que se generan con C++ Builder, ya que la VCL llama a *CoInitializeEx* automáticamente al iniciar cualquier aplicación que utilice COM.

Un servidor de bloqueos

En determinadas aplicaciones, es inaceptable que un usuario se enfrasque en una larga operación de modificación de un registro ... para descubrir en el momento de la grabación que no puede guardar sus cambios, porque otro usuario ha modificado mientras tanto el mismo registro. Esto es lo que sucede habitualmente en las aplicaciones desarrolladas para bases de datos cliente/servidor.

A lo largo de este libro hemos visto técnicas que ayudan a disminuir la cantidad de conflictos que se pueden producir. Ahora ofreceremos una alternativa radical: implementaremos una aplicación que actúe como servidor remoto de bloqueos. La aplicación se ejecutará en un punto central de la red, que no tiene por qué coincidir con el del servidor SQL. Cada vez que un usuario intente acceder a un registro para modificarlo, debe pedir permiso al servidor, para ver si algún otro usuario está utilizándolo. El servidor mantendrá una lista de los registros “bloqueados”. Y cuando una aplicación a la que se le haya concedido el derecho a editar un registro termine su edición, ya sea cancelando o grabando, la aplicación debe retirar el bloqueo mediante otra llamada al servidor.

Primero desarrollaremos el servidor, para después ponerlo a punto con una aplicación de prueba. De acuerdo a lo explicado anteriormente, son solamente dos los procedimientos que tendremos que implementar para el servidor de bloqueos:

```
HRESULT LockRow(BSTR TableName, int Key);
HRESULT UnlockRow(BSTR TableName, int Key);
```

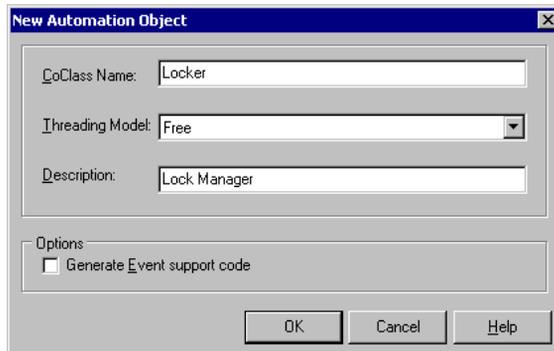
Las funciones devolverán los valores *S_OK*, para indicar éxito, y *S_FALSE* cuando fracasan. Luego el cliente deberá transformar estos códigos en excepciones. He decidido, para simplificar el código del servidor, que las claves primarias sean siempre un número entero. Según hemos visto en los ejemplos de este libro, existen razones convincentes para introducir claves artificiales incluso en los casos en que semánti-

camente la clave primaria natural sería una clave compuesta. El lector puede, no obstante, aumentar las posibilidades del servidor añadiendo métodos que bloqueen registros de tablas con otros tipos de claves primarias.

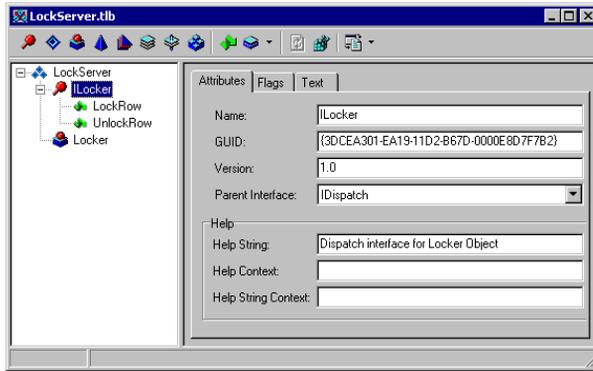
De modo que iniciamos una nueva aplicación (*File | New application*). A la ventana principal le damos el nombre de *wndMain* (¡qué original!). Más adelante, podrá modificar esta ventana para que aparezca como un icono en la bandeja de iconos de Windows. En el capítulo 5 ya hemos explicado cómo hacerlo. Incluso, si solamente va a utilizar el servidor en Windows NT, puede transformar esta aplicación en un servicio.

Guardamos ahora la aplicación. Si lo desea, llame *Main* a la unidad de la ventana principal. Ahora bien, le ruego que llame *LockServer* al fichero de proyecto, y que no mueva durante el desarrollo el directorio donde estará situado. A continuación, ejecute el diálogo del Depósito de Objetos (*File | New*), vaya a la segunda página (*ActiveX*), y pulse el icono *Automation Object*.

En el diálogo de creación de objetos de automatización, teclee *Locker* como nombre de clase, y deje el modo de generación de instancias como *Multiple instance*, esto quiere decir que una sola instancia de la aplicación podrá ejecutar concurrentemente el código de múltiples instancias de objetos de automatización. En el modelo de concurrencia especificaremos *Free*. Cuando cierre el diálogo, guarde el proyecto y nombre a la nueva unidad generada como *LockImpl*.



Aparecerá entonces el Editor de la Biblioteca de Tipos del proyecto. Utilizando el botón *Method* debemos crear el par de métodos *LockRow* y *UnlockRow* bajo el nodo correspondiente a la interfaz *ILocker*, con los prototipos establecidos al principio de esta sección. Después debemos pulsar el botón *Refresh*, para que los cambios se reflejen en la unidad de declaración de las interfaces, *LockServer_TLB*, y en la unidad *LockerImpl*, en la cual implementaremos los métodos definidos.



La implementación de la lista de bloqueos

La estructura de datos a elegir para implementar la lista de bloqueos dependerá en gran medida de la estimación que hagamos del uso del servidor. Mis cálculos consisten en que un número relativamente pequeño de tablas contendrán un número elevado de bloqueos. Por lo tanto, mi sugerencia es utilizar una lista vectorial de nombres de tablas, ordenada alfabéticamente, y asociar a cada una de las entradas en esta lista, un puntero a un árbol binario ordenado.

Debemos crear una nueva unidad, con el nombre *trees*, que implemente las operaciones de borrado y de búsqueda e inserción en un árbol binario. Este es el fichero de cabecera de la unidad:

```
//-----
#ifndef TreesH
#define TreesH
//-----

struct TTree
{
    int Key;
    TTree *Left;
    TTree *Right;
};

typedef TTree *PTree;

bool TreeInsert(int AKey, PTree &ATree);
bool TreeDelete(int AKey, PTree &ATree);

#endif
```

El procedimiento *TreeInsert* trata de insertar un nuevo nodo con la clave indicada, y devuelve *False* cuando ya existe un nodo con esa clave. *TreeDelete*, por su parte, busca el nodo con la clave especificada para borrarlo, y devuelve *True* sólo cuando lo ha podido encontrar.

```

#include "Trees.h"

bool TreeInsert(int AKey, PTree &ATree)
{
    PTree *T = &ATree;
    while (*T)
        if (AKey == (*T)->Key)
            return False;
        else if (AKey < (*T)->Key)
            T = &(*T)->Left;
        else
            T = &(*T)->Right;
    *T = new TTree;
    (*T)->Key = AKey;
    (*T)->Left = NULL;
    (*T)->Right = NULL;
    return True;
}

bool TreeDelete(int AKey, PTree &ATree)
{
    PTree *T = &ATree;
    while (*T)
    {
        if (AKey == (*T)->Key)
        {
            PTree Del = *T;
            if ((*T)->Left == NULL)
            {
                *T = Del->Right;
                delete Del;
            }
            else if ((*T)->Right == NULL)
            {
                *T = Del->Left;
                delete Del;
            }
            else
            {
                // Buscar el mayor del subárbol izquierdo
                PTree Mayor = Del->Left;
                while (Mayor->Right != NULL)
                    Mayor = Mayor->Right;
                Del->Key = Mayor->Key;
                TreeDelete(Mayor->Key, Del->Left);
            }
            return True;
        }
        if (AKey < (*T)->Key)
            T = &(*T)->Left;
    }
}

```

```

        else
            T = &(*T)->Right;
    }
    return False;
}

```

La explicación de esta unidad va más allá de los propósitos del libro. La función de inserción es relativamente sencilla. La de borrado es ligeramente más complicada; el caso más barroco sucede cuando se intenta borrar un nodo que tiene ambos hijos asignados. Entonces hay que buscar el mayor de los nodos menores que el que se va a borrar (es decir, el nodo anterior en la secuencia de ordenación). Se mueve el valor de este nodo al nodo que se iba a borrar y se elimina físicamente el antiguo nodo, del que se ha extraído el valor anterior.

Control de concurrencia

La implementación directa de los métodos *LockRow* y *UnlockRow* tiene lugar en la unidad *LockImpl*, que es la que contiene la declaración de la clase *TLockerImpl*. Pero antes crearemos una clase *TLockManager*, a partir de la cual crearemos una instancia global al servidor. El objeto global nos servirá para almacenar la lista de tablas que contienen bloqueos, el número de bloqueos existente en cualquier instante y un objeto de clase *TCriticalSection*, que evitará colisiones en el uso de los objetos anteriores. Todos los objetos de automatización trabajarán sobre la instancia global.

La declaración de la clase la podemos ubicar dentro del propio fichero *trees.h*:

```

class TLockManager
{
protected:
    TStringList* fTables;
    int fLocks;
    TCriticalSection* cSection;
public:
    TLockManager();
    ~TLockManager();
    HRESULT LockRow(BSTR TableName, int AKey);
    HRESULT UnlockRow(BSTR TableName, int AKey);

    static TLockManager *lockManager;
};

```

La estructura de la lista de bloqueos es la siguiente: en el primer nivel tendremos una lista ordenada de cadenas, almacenada en la variable *fTables*. Para cada tabla guardaremos un árbol binario diferente, en el vector paralelo *Objects* de la clase *TStringList*. Por su parte, la clase *TCriticalSection* está definida dentro de la unidad *SyncObjs* de la VCL 4. Las *secciones críticas* son uno de los mecanismos que ofrece Windows para sincronizar procesos paralelos. Operan dentro de una misma aplicación, a diferencia

de otros recursos de sincronización globales como los semáforos y *mutexes*, pero son más eficientes.

He aquí el constructor y el destructor de la clase *TLockManager*:

```
TLockManager::TLockManager() :
    fTables(new TStringList),
    cSection(new TCriticalSection),
    fLocks(0)
{
    fTables->Sorted = True;
    fTables->Duplicates = dupIgnore;
}

TLockManager::~TLockManager()
{
    delete cSection;
    delete fTables;
}

TLockManager *TLockManager::lockManager = 0;
```

Los métodos públicos se implementan dentro de la protección de la sección crítica, de modo que dos objetos servidores no puedan simultáneamente realizar cambios en la lista de tablas o en los árboles binarios asociados. Cada cliente que se conecte al servidor de bloqueos lanzará un hilo independiente. Si *Enter* encuentra que otro hilo ha ejecutado *Enter* y todavía no ha salido con *Leave*, pone en espera al hilo que lo ha ejecutado, hasta que quede liberada la sección:

```
HRESULT TLockManager::LockRow(BSTR TableName, int AKey)
{
    cSection->Enter();
    try
    {
        int i = fTables->Add(TableName);
        PTree t = PTree(fTables->Objects[i]);
        if (! TreeInsert(AKey, t))
            return S_FALSE;
        fTables->Objects[i] = (TObject*) t;
        fLocks++;
    }
    finally
    {
        cSection->Leave();
    }
    return S_OK;
}

HRESULT TLockManager::UnlockRow(BSTR TableName, int AKey)
{
    cSection->Enter();
    try
    {
        int i = fTables->IndexOf(TableName);
```

```

        if (i != -1)
        {
            PTree t = PTree(fTables->Objects[i]);
            if (TreeDelete(AKey, t))
            {
                fTables->Objects[i] = (TObject*) t;
                fLocks--;
                return S_OK;
            }
        }
    }
__finally
{
    cSection->Leave();
}
return S_FALSE;
}

```

La construcción y destrucción de la instancia global se realizará en el código de inicio del servidor:

```

WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
    TLockManager::lockManager = new TLockManager;
    try
    {
        Application->Initialize();
        Application->CreateForm(__classid(TwndMain), &wndMain);
        Application->Run();
    }
    catch (Exception &exception)
    {
        delete TLockManager::lockManager;
        Application->ShowException(&exception);
    }
    return 0;
}

```

Finalmente estamos listos para implementar los métodos de automatización. La inserción y eliminación de un registro en la lista de bloqueos se realiza del siguiente modo:

```

STDMETHODIMP TLockerImpl::LockRow(BSTR TableName, int AKey)
{
    return TLockManager::lockManager->LockRow(TableName, AKey);
}

STDMETHODIMP TLockerImpl::UnlockRow(BSTR TableName, int AKey)
{
    return TLockManager::lockManager->UnlockRow(TableName, AKey);
}

```

Poniendo a prueba el servidor

Ahora que hemos completado la aplicación servidora, estamos en condiciones de ponerla a prueba. Comenzaremos las pruebas en modo local, con el servidor y los clientes situados en el mismo ordenador; así evitaremos los problemas iniciales que pueden surgir durante la comunicación remota entre aplicaciones. El primer paso es compilar y ejecutar el servidor, al menos una vez, para que éste grabe las entradas necesarias en el registro de Windows.

A continuación creamos una nueva aplicación, en la cual colocamos una tabla enlazada a alguna base de datos SQL; el alias *IBLOCAL* de los ejemplos de C++ Builder puede servir. Añada es este proyecto los ficheros *LockerServer_TLB* con extensiones *cpp* y *h*, que contienen las declaraciones en C++ de la biblioteca de tipos. De este modo, podremos utilizar las interfaces duales en vez de tener que utilizar variantes engorrosos.

Para concretar, supongamos que la tabla elegida es *employee*, cuya clave primaria es la columna *Emp_No*. La interfaz de usuario, en sí, será muy sencilla: una rejilla de datos y una barra de navegación bastan. Vamos a la declaración de la clase *TForm1*, a la sección **private**, y añadimos la siguiente declaración:

```
class TForm1 : public TForm
{
    // ...
private:
    TCOMILocker Locker;
};
```

La clase *TCOMILocker* corresponde a la interfaz “inteligente” declarada en la biblioteca de tipos importada. La variable se inicializa durante la construcción del formulario; no será necesario destruir el enlace explícitamente, pues de eso se encargará el destructor de la clase de la ventana principal:

```
__fastcall TForm1::TForm1(TComponent *Owner)
: TForm(Owner), Locker(CoLocker::Create())
{
}
```

El resto del código se concentrará en los eventos de transición de estado del componente de acceso a la tabla. Estos serán los eventos necesarios:

Acción	Eventos
Bloquear	<i>BeforeEdit</i>
Desbloquear	<i>OnEditError, AfterPost, AfterCancel</i>

En este caso simple, en que no necesitamos más instrucciones dentro de los manejadores de estos eventos, podemos crear tres métodos para que sirvan de receptores:

```

void __fastcall TForm1::Table1BeforeEdit(TDataSet *DataSet)
{
    WideString tableName = Table1->TableName;
    HRESULT hr = Locker.LockRow(tableName, Table1EMP_NO->AsInteger);
    OleCheck(hr);
    if (hr == S_FALSE)
        throw Exception("Registro bloqueado");
}

void __fastcall TForm1::Table1AfterPostCancel(TDataSet *DataSet)
{
    // Compartido por AfterPost y AfterCancel
    WideString tableName = Table1->TableName;
    OleCheck(Locker.UnlockRow(tableName, Table1EMP_NO->AsInteger));
}

void __fastcall TForm1::Table1EditError(TDataSet *DataSet,
    EDatabaseError *E, TDataAction &Action)
{
    WideString tableName = Table1->TableName;
    OleCheck(Locker.UnlockRow(tableName, Table1EMP_NO->AsInteger));
}

```

Hemos necesitado la variable temporal de tipo *WideString* para crear el puntero de tipo *BSTR* que debemos pasar a los métodos del administrador de bloqueos. *BSTR* se representa como una cadena Unicode con un prefijo de cuatro bytes con la longitud. Si no hubiéramos recurrido a esta clase importada desde Delphi, tendríamos que llamar a las funciones del API de Windows *SysAllocString* y *SysFreeString*, además de garantizar la llamada de esta última en caso de excepción.

Ahora puede ejecutar dos copias de la aplicación y comprobar que, efectivamente, cuando editamos una fila de la tabla de empleados bloqueamos el acceso a la misma desde la otra copia de la aplicación.

Para sustituir el servidor local por un servidor remoto sólo necesitamos cambiar el mecanismo de creación de la instancia *Locker*:

```

__fastcall TForm1::TForm1(TComponent *Owner)
: TForm(Owner), Locker(CoLocker::CreateRemote("NombreServidor"))
{
}

```

EN ESTE CAPÍTULO NOS OCUPAMOS del modelo de desarrollo de aplicaciones en múltiples capas, y en particular de los servicios que ofrece la tecnología Midas de Inprise (o Borland, como prefiera). Enfocaremos el desarrollo de servidores de datos en la capa intermedia, y veremos qué componentes nos proporciona C++ Builder para desarrollar las partes del servidor y del cliente de la aplicación, estudiando los mecanismos particulares de este modelo para resolver los problemas que ocasiona el acceso concurrente.

¿Qué es Midas?

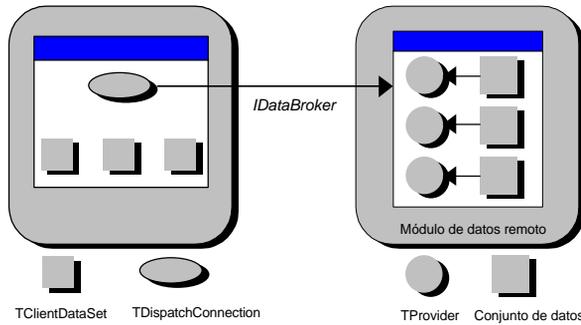
Las siglas Midas quieren decir, con un poco de buena voluntad, *Multi-tiered Distributed Application Services*, que traducido viene a ser, poco más o menos, Servicios para Aplicaciones Distribuidas en Múltiples Capas. Midas no es una aplicación, ni un componente, sino una serie de servicios o mecanismos que permiten transmitir conjuntos de datos entre dos aplicaciones. En la primera versión de Midas, que apareció con Delphi 3.0, el vehículo de transmisión era DCOM, aunque podíamos utilizar un sustituto, también basado en COM, denominado OLEEnterprise. Con C++ Builder 4 podemos transmitir conjuntos de datos entre aplicaciones utilizando indistintamente COM/DCOM, OLEEnterprise y TCP/IP.

En el mecanismo básico de comunicación mediante Midas intervienen dos aplicaciones. Una actúa como servidora de datos y la otra actúa como cliente. Lo normal es que ambas aplicaciones estén situadas en diferentes ordenadores, aunque en ocasiones es conveniente que estén en la misma máquina, como veremos más adelante. También es habitual que el servidor sea un ejecutable, aunque si vamos a colocar el cliente y el servidor en el mismo puesto es posible, y preferible, programar un servidor DLL dentro del proceso.

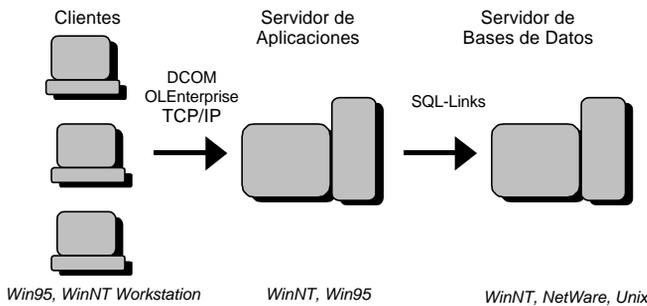
A grandes rasgos, la comunicación cliente/servidor se establece a través de una interfaz COM nombrada *IDataBroker*, que es una interfaz de automatización. Esta interfaz permite el acceso a un conjunto de *proveedores*; cada proveedor es un objeto que soporta la interfaz *IProvider*. Los proveedores se sitúan en la aplicación servidora, y cada

uno de ellos proporciona acceso al contenido de un conjunto de datos diferente. Esta estructura tiene su paralelo en la parte cliente. Los componentes derivados de la clase abstracta *TDispatchConnection* utilizan la interfaz *IDataBroker* de la aplicación servidora, y ponen a disposición del resto de la aplicación cliente los punteros a las interfaces *IProvider*. Cada clase concreta derivada de *TDispatchConnection* implementa la comunicación mediante un protocolo determinado: *TDCOMConnection*, *TSocketConnection* y *TOLEEnterpriseConnection*.

Nuestro viejo conocido, el componente *TClientDataSet*, puede utilizar una interfaz *IProvider* extraída de un *TDispatchConnection* como fuente de datos. A partir de estos componentes, la arquitectura de la aplicación es similar a la tradicional, con conjuntos de datos basados en el BDE. El siguiente esquema muestra los detalles de la comunicación entre los clientes y el servidor de aplicaciones:



Las más popular de las posibles configuraciones de un sistema de este tipo es la clásica aplicación en tres capas, cuya estructura se muestra en el siguiente diagrama:



En esta configuración los datos se almacenan en un servidor dedicado de bases de datos. El sistema operativo que se ejecuta en este ordenador no tiene por qué ser Windows: UNIX, en cualquiera de sus mutaciones, o NetWare pueden ser alternativas mejores, pues la concurrencia está mejor diseñada y, en mi opinión, son más estables, aunque también más difíciles de administrar correctamente. Hay un segundo

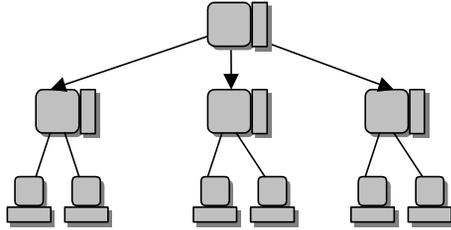
ordenador, el servidor de aplicaciones, que actúa de capa intermedia. En esta máquina está instalado el BDE y los SQL Links, con el propósito de acceder a los datos del servidor SQL. El sistema operativo, por lo tanto, debe ser Windows NT Server, preferentemente, ó Windows NT Workstation e incluso Windows 95/98. La aplicación escrita en C++ Builder (o incluso Delphi) que se ejecuta aquí no tiene necesidad de presentar una interfaz visual; puede ejecutarse en segundo plano, aunque es conveniente disponer de algún monitor de control. Finalmente, los ordenadores clientes son los que se encargan de la interfaz visual con los datos; éstos son terminales relativamente baratas, que ejecutan preferentemente Windows 95/98 ó NT Workstation. En estos ordenadores no se instala el Motor de Datos de Borland, pues la comunicación entre ellos y el servidor intermedio se realiza a través de DCOM, OLEEnterprise o TCP/IP.

Cuándo utilizar y cuándo no utilizar Midas

Las bondades de Midas, pero sobre todo la propaganda acerca de la nueva técnica, ha llevado a muchos equipos de programación a lanzarse indiscriminadamente al desarrollo utilizando el modelo que acabo de presentar. Desde mi punto de vista, muchas de las aplicaciones planteadas no justifican el uso de este modelo. De lo que se trata no es de la conveniencia indiscutible de estratificar los distintos niveles de tratamiento de datos en una aplicación de bases de datos, sino de si es rentable o no que esta división se exprese físicamente. Es algo de sentido común el hecho de que al añadir una capa adicional de software o de hardware, cuya única función es la de servir de correa de transmisión, solamente logramos ralentizar la ejecución de la aplicación.

Olvidémonos por un momento de la estratificación metodológica y concentrémonos en el análisis de la eficiencia. ¿Cuál es la ventaja del modelo de dos capas, más conocido como modelo cliente/servidor? La principal es que gran parte de las reglas de empresa pueden implementarse en el ordenador que almacena los datos. Por lo tanto, para su evaluación los datos no necesitan viajar por la red hasta alcanzar el nodo en que residen las reglas. ¿Qué sucede cuando se añade una capa intermedia? Pues que en la mayoría de las aplicaciones no existen reglas de empresa lo suficientemente complejas como para justificar un nivel intermedio: casi todo lo que puede hacer un servidor Midas puede implementarse en el propio servidor SQL. Existe una excepción importante para este razonamiento: si nuestra aplicación requiere reglas de empresa que involucren simultáneamente a varias bases de datos. Por ejemplo, el inventario de nuestra empresa reside en una base de datos Oracle, pero el sistema de facturación utiliza InterBase. Ninguno de los servidores SQL puede asumir por sí mismo la ejecución de las reglas correspondientes, por lo que la responsabilidad debe descargarse en un servidor Midas.

Existe, sin embargo, una técnica conocida como *balance de carga*, que en C++ Builder 3 solamente podía implementarse con OLEEnterprise, pero que en la versión 4 también puede aplicarse a otros protocolos. La técnica consiste en disponer de una batería de servidores de capa intermedia similares, que ejecuten la misma aplicación servidora y que se conecten al mismo servidor SQL. Los clientes, o estaciones de trabajo, se conectan a estos servidores de forma balanceada, de forma tal que cada servidor de capa intermedia proporcione datos aproximadamente al mismo número de clientes.



Se deben dar dos condiciones para que esta configuración disminuya el tráfico de red y permita una mayor eficiencia:

- El segmento de red que comunica al servidor SQL con los servidores Midas y el que comunica a los servidores Midas con los clientes deben ser diferentes físicamente.
- Los servidores Midas deben asumir parte del procesamiento de las reglas de empresa, liberando del mismo al servidor SQL.

Otra desventaja del uso de Midas consiste en que las aplicaciones clientes deben emplear conjuntos de datos clientes, y estos componentes siguen una filosofía optimista respecto al control de cambios. Ya es bastante difícil convencer a un usuario típico de las ventajas de los bloqueos optimistas, como para además llevar este modo de acción a su mayor grado. Los conjuntos de datos clientes obligan a realizar las grabaciones mediante una acción explícita (*ApplyUpdates*), y es bastante complicado disfrazar esta acción de modo que pase desapercibida para el usuario. Es más, le sugiero que se olvide de las rejillas de datos en este tipo de aplicación. Como ya sabe, si la curiosidad del usuario le impulsa a examinar el último registro de una tabla, no pasa nada pues el BDE implementa eficientemente esta operación. Pero un conjunto de datos clientes, al igual que una consulta, se ve obligado a traer todos los registros desde su servidor.

De todos modos, existen otras consideraciones aparte de la eficiencia que pueden llevarnos a utilizar servidores de capa intermedia. La principal es que los ordenadores en los que se ejecutan las aplicaciones clientes no necesitan el Motor de Datos de Borland ni, lo que es más importante, el cliente del sistema cliente/servidor. Co-

nozco dos importantes empresas de *tele-marketing* que programan en Delphi y C++ Builder; una de ellas tiene sus datos en Oracle y la otra en Informix. Es bastante frecuente que, tras contratar una campaña, tengan que montar de un día para otro treinta o cuarenta ordenadores para los nuevos operadores telefónicos y, después de instalar el sistema operativo, se vean obligados a instalar y configurar el BDE y el cliente de Oracle e Informix en cada uno de ellos. Evidentemente, todo resulta ser más fácil con los clientes de Midas, que solamente necesitan para su funcionamiento una DLL de 206 KB (C++ Builder 4).

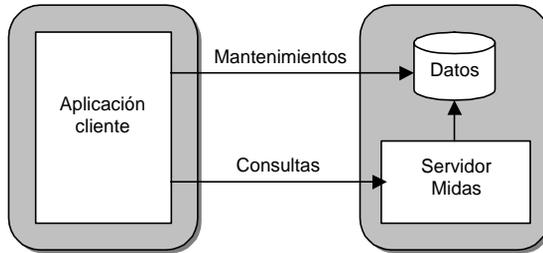
Midas y las bases de datos de escritorio

Los razonamientos anteriores suponen que nuestra base de datos reside en un sistema cliente/servidor. Curiosamente, una aplicación que funcione con Paradox o dBase puede beneficiarse mucho de utilizar un servidor Midas para determinadas tareas. No voy a sugerirle que utilice conjuntos de datos clientes para el mantenimiento de las tablas, pues estas operaciones se realizan más eficientemente accediendo directamente a los ficheros. Pero las consultas sobre la base de datos son excelentes candidatas a ser sustituidas por componentes *TClientDataSet*. Son las once de la mañana, y todos en la oficina se afanan en introducir datos de pedidos desde sus ordenadores a la Gran Base de Datos Central (que a pesar de las mayúsculas está en formato Paradox). A esa hora el Gran Jefe se digna en honrar con su presencia a sus empleados. Se repantinga en su sillón de cuero, enciende su Gran Ordenador y ejecuta el único comando que conoce de la aplicación: obtener un gráfico con la distribución de las ventas. Pero mientras el gráfico se genera, el rendimiento de la red cae en picado, las aplicaciones parecen moverse en cámara lenta, y los pobres empleados miran al cielo raso, suspiran y calculan cuántos meses quedan para las próximas vacaciones.

Lo que ha sucedido es que los datos del gráfico se reciben de un *TQuery*, mediante una consulta que implica el encuentro entre cinco tablas, un **group by** y una ordenación posterior. Y el encargado de ejecutar esa consulta es el intérprete local de SQL del Gran Ordenador. Esto significa que todos los datos de las tablas implicadas deben viajar desde la Gran Base de Datos Central hacia la máquina del Gran Jefe, y consumir el recurso más preciado: el ancho de banda de la red.

Mi propuesta es la siguiente: que las operaciones de mantenimiento sigan efectuándose como hasta el momento, accediendo mediante componentes *TTable* a la base de datos central. Pero voy a instalar un servidor Midas en el ordenador de la Gran Base de Datos, que contendrá la consulta que utiliza el gráfico, y exportará ese conjunto de datos. Por otra parte, cuando la aplicación necesite imprimir el gráfico, extraerá sus datos de un conjunto de datos clientes que se debe conectar al servidor Midas. ¿Dónde ocurrirá la evaluación de la consulta? Está claro que en la misma máquina que contiene la base de datos y, como consecuencia, los datos originales no tendrán

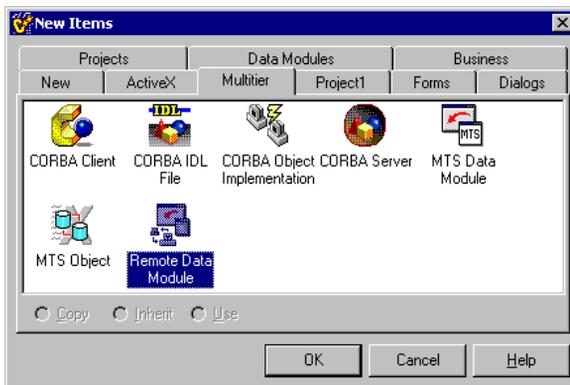
que viajar por la red. El conjunto resultado es normalmente pequeño, y será lo único que tendrá que desplazarse hasta el ordenador del Gran Jefe. ¿Resultado? Menos tráfico en la red (y empleados que miran menos al techo).



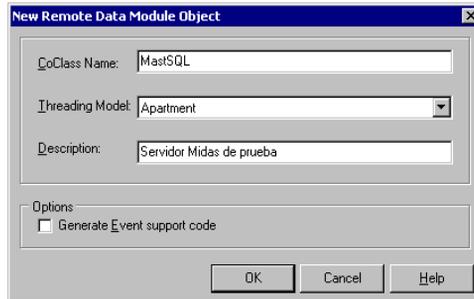
Un servidor Midas también puede encargarse de ejecutar operaciones en lote que no requieran interacción con el usuario. Estas operaciones se activarían mediante una petición emitida por un cliente, y se ejecutarían en el espacio de procesos del servidor. De esta forma, estaríamos implementando algo parecido a los procedimientos almacenados, pero sobre una base de datos de escritorio.

Módulos de datos remotos

Una vez sopesados los pros y los contras de las aplicaciones Midas, veamos como crearlas. Para desarrollar una aplicación en tres capas es necesario, en primer lugar, crear la aplicación intermedia que actuará como servidor de aplicaciones. De esta manera, la aplicación cliente conocerá la estructura de los datos con que va a trabajar. Es parecido a lo que ocurre en las aplicaciones en una o dos capas: necesitamos que las tablas estén creadas, y de ser posible con datos de prueba, para poder trabajar sobre su estructura.



Partiremos de una aplicación vacía; en principio no necesitamos el formulario principal, pero lo dejaremos con el propósito de controlar la ejecución del servidor. Primero creamos un módulo de datos remoto, por medio del Depósito de Objetos. El icono necesario se encuentra en la página *Multitier* del Depósito, bajo el castizo nombre de *Remote data module*. Al seleccionar el icono y pulsar el botón *Ok*, C++ Builder nos pide un nombre de clase y un modelo de concurrencia, al igual que para los objetos de automatización que hemos visto en el capítulo anterior:



Como nombre de clase utilizaremos *MastSQL*; guardaremos la aplicación como *Servidor.Apl*, por lo que el identificador de clase que utilizaremos más adelante para referirnos al objeto de automatización será la concatenación de estos nombres: *Servidor.Apl.MastSQL*. Como modelo de instanciación, utilizaremos *Multiple instances*, que es el utilizado por omisión por la ATL; recuerde que el modelo de instanciación puede cambiarse en la página *ATL* de las opciones del proyecto. De esta forma, cuando se conecten varios clientes al servidor, la aplicación se ejecutará una sola vez. Por cada cliente, sin embargo, habrá un módulo remoto diferente, ejecutándose en su propio hilo. Y como estamos creando un servidor remoto, podemos ignorar el modelo de concurrencia.

A primera vista, el módulo incorporado al proyecto tiene el mismo aspecto que un módulo normal, pero no es así. Si abrimos el Administrador de Proyectos, veremos que C++ Builder ha incluido una biblioteca de tipos en el proyecto. Esto se debe a que el módulo de datos remoto implementa una interfaz, denominada *IDataBroker*, que Borland define como descendiente de la interfaz de automatización *IDispatch*. En la biblioteca se define una nueva interfaz y una clase de componentes asociada:

```
[
    uuid(03B79CE1-F03A-11D2-B67D-0000E8D7F7B2),
    version(1.0),
    helpstring("Dispatch interface for MastSQL Object"),
    dual,
    oleautomation
]
interface IMastSQL: IDataBroker
{
};
```

```

[
    uuid(03B79CE3-F03A-11D2-B67D-0000E8D7F7B2),
    version(1.0),
    helpstring("MastSQL Object")
]
coclass MastSQL
{
    [default] interface IMastSQL;
};

```

Recuerde que no debemos modificar directamente los ficheros importados a partir de la biblioteca, sino a través del Editor de la Biblioteca de Tipos.

Para controlar el número de clientes que se conectan al servidor, aprovecharemos que por cada cliente se crea un nuevo objeto de tipo *TMastSQL*. En el formulario principal añadimos un componente *Label1*, un nuevo atributo de tipo *TCriticalSection* y un método *ReportConnection*:

```

class TForm1 : public TForm
{
    __published:    // IDE-managed Components
    TLabel *Label1;
private:         // User declarations
    TCriticalSection *csection;
    int fConnections;
public:          // User declarations
    __fastcall TForm1(TComponent* Owner);
    void __fastcall ReportConnection(int delta);
};

```

La sección crítica se crea y se destruye explícitamente en el constructor del formulario, y durante la respuesta a *OnClose*, respectivamente:

```

__fastcall TForm1::TForm1(TComponent* Owner) : TForm(Owner)
{
    csection = new TCriticalSection;
}

void __fastcall TForm1::FormClose(TObject *Sender,
    TCloseAction &Action)
{
    delete csection;
}

```

La implementación de *ReportConnection* es la siguiente:

```

void __fastcall TForm1::ReportConnection(int delta)
{
    csection->Enter();
    try
    {
        fConnections += delta;
    }
}

```

```

        Labell->Caption = IntToStr(fConnections) +
            " clientes conectados";
    }
    finally
    {
        csection->Leave();
    }
}

```

Regresamos al módulo de datos, incluimos la cabecera de la unidad principal y creamos estos métodos como respuesta a los eventos *OnCreate* y *OnDestroy* del módulo remoto:

```

void __fastcall TMastSQL::MastSQLCreate(TObject* Sender)
{
    Form1->ReportConnection(+1);
}

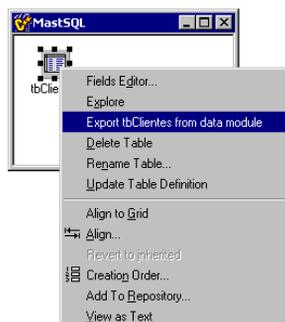
void __fastcall TMastSQL::MastSQLDestroy(TObject* Sender)
{
    Form1->ReportConnection(-1);
}

```

Proveedores

Las aplicaciones clientes deben extraer sus datos mediante una interfaz remota cuyo nombre es *IProvider*. El propósito del módulo de datos remoto, además de contener los componentes de acceso a datos, es exportar un conjunto de proveedores por medio de la interfaz *IDataBroker*. Por lo tanto, nuestro próximo paso es crear estos proveedores. Comenzaremos con el caso más sencillo: una sola tabla. Más adelante veremos como manejar relaciones *master/detail*, y cómo pueden manejarse transacciones de forma explícita.

Colocamos una tabla en el módulo de datos remotos, cambiamos su nombre a *tbClientes*, su base de datos a *bdemos* y su nombre de tabla a *customer.db*; después la abrimos mediante su propiedad *Active*. Pulsamos el botón derecho del ratón sobre la tabla y elegimos el comando *Export tbClientes from data module*.



Con esta acción hemos provocado que una nueva propiedad, *tbClientes* de sólo lectura, aparezca en la interfaz del objeto de automatización. La declaración IDL de la interfaz se ha modificado de la siguiente forma:

```
[
    uuid(03B79CE1-F03A-11D2-B67D-0000E8D7F7B2),
    version(1.0),
    helpstring("Dispatch interface for MastSQL Object"),
    dual,
    oleautomation
]
interface IMastSQL: IDataBroker
{
    [propget, id(0x00000001)]
    HRESULT _stdcall tbClientes([out, retval] IProvider ** Value);
};
```

Como se trata de una interfaz sólo lectura, se implementa mediante un método de nombre *get_tbClientes*, generado automáticamente por C++ Builder:

```
STDMETHODIMP TMastSQLImpl::get_tbClientes(IProvider** Value)
{
    try
    {
        _di_IProvider IProv = m_DataModule->tbClientes->Provider;
        IProv->AddRef();
        *Value = IProv;
    }
    catch(Exception &e)
    {
        return Error(e.Message.c_str(), IID_IMastSQL);
    }
    return S_OK;
}
```

En primer lugar, observe que la implementación de la interfaz está a cargo de la clase *TMastSQLImpl*, mientras que el módulo de datos está asociado a la clase *TMastSQL*. No importa: la primera clase desciende por herencia de la segunda, y es la que será utilizada por la fábrica de clases para la creación de objetos.

Los conjuntos de datos basados en el BDE tienen una propiedad *Provider*, del tipo *IProvider*, que ha hecho posible exportar directamente sus datos desde el módulo remoto. Ahora bien, es preferible realizar la exportación incluyendo explícitamente un componente *TProvider*, de la página *Data Access* de la Paleta de Componentes, conectándolo a un conjunto de datos y exportándolo mediante su menú de contexto. La ventaja de utilizar este componente es la posibilidad de configurar opciones y de crear manejadores para los eventos que ofrece, ganando control en la comunicación entre el cliente, el servidor de aplicaciones y el servidor de bases de datos.

Para deshacer la exportación de *tbClientes*, debemos buscar en primer lugar la Biblioteca de Tipos (*View | Type library*), seleccionar la propiedad *tbClientes* de la interfaz

IMastSQL y eliminarla. Después, en la unidad del módulo de datos, hay que eliminar la función *get_tbClientes*. Traemos entonces un componente *TProvider*, de la página *Midas*, y lo situamos sobre el módulo de datos. Cambiamos las siguientes propiedades:

Propiedad	Valor
<i>Name</i>	<i>Clientes</i>
<i>DataSet</i>	<i>tbClientes</i>
<i>Options</i>	Añadir <i>poIncFieldProps</i>

La opción *poIncFieldProps* hace que el proveedor incluya en los paquetes que envía al cliente información sobre propiedades de los campos, tales como *DisplayLabel*, *DisplayFormat*, *Alignment*, etc. De esta forma, podemos realizar la configuración de los campos en el servidor y ahorrarnos repetir esta operación en sus clientes. Más adelante estudiaremos otras propiedades de *TProvider*, además de sus eventos. Finalmente, pulsamos el botón derecho del ratón sobre el componente y ejecutamos el comando *Export Clientes from data module*.

Como estamos utilizando una tabla perteneciente a una base de datos de escritorio, es necesario activar la opción *ResolveToDataSet* del *TProvider*. Si esta opción no está activa, el proveedor genera sentencias SQL que envía directamente a la base de datos, saltando por encima del conjunto de datos asociado. Esto es lo más conveniente para las bases de datos SQL a las cuales se accede por medio del BDE, pero no vale para Paradox, dBase y Access.

Una vez que hemos exportado una interfaz *IProvider*, de una forma u otra, podemos guardar el proyecto y ejecutarlo la primera vez; así se registra el objeto de automatización dentro del sistema operativo. Recuerde que para eliminar las entradas del registro, se debe ejecutar la aplicación con el parámetro */unregserver* en la línea de comandos.

Servidores remotos y conjuntos de datos clientes

Es el momento de programar la aplicación cliente. Si el lector no dispone de una red con DCOM configurado de algún modo, no se preocupe, porque podemos probar el cliente y el servidor dentro de la misma máquina. Iniciamos una aplicación nueva, con un formulario vacío, y añadimos un módulo de datos de los de siempre. Sobre este módulo de datos colocamos un componente *TDCOMConnection*, de la página *Midas*, cuyas propiedades configuramos de este modo:

Propiedad	Valor	Significado
<i>Computer</i>		Nombre del ordenador donde reside el servidor. Se deja vacío si es local.

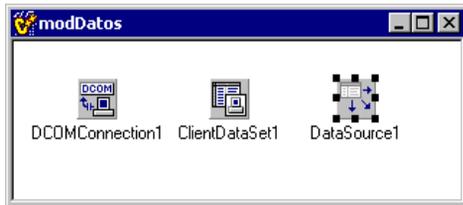
Propiedad	Valor	Significado
<i>ServerName</i>	<i>Servidor.Apl.MastSQL</i>	Identificador de clase del objeto de automatización
<i>ServerGUID</i>		Lo llena automáticamente C++ Builder
<i>Connected</i>	<i>True</i>	Al activarla, se ejecuta el servidor de aplicaciones

Por cada proveedor que exporte el módulo de datos remoto y que nos interese, traemos al módulo local un componente *TClientDataSet*, de la página *Midas* de la Paleta de Componentes. Ya hemos visto en funcionamiento a este componente en el capítulo 32, pero trabajando con datos extraídos de ficheros locales. Ahora veremos qué propiedades, métodos y eventos tenemos que utilizar para que funcione con una interfaz *IProvider* como origen de datos. En concreto, para este proyecto traemos un *TClientDataSet* al módulo local, y asignamos las siguientes propiedades; después traemos un *TDataSource* para poder visualizar los datos.

Propiedad	Valor
<i>RemoteServer</i>	<i>DCOMConnection1</i>
<i>ProviderName</i>	<i>Cientes</i>
<i>Active</i>	<i>True</i>

Hay un par de propiedades que controlan la forma en que el conjunto de datos cliente extrae los registros del servidor de aplicaciones. *FetchOnDemand*, por ejemplo, debe ser *True* (valor por omisión) para que los registros se lean cuando sea necesario; si es *False*, hay que llamar explícitamente al método *GetNextPacket*, por lo que se recomienda dejar activa esta propiedad. *PacketRecords* indica el número de registros que se transfiere en cada pedido. En el caso en que es -1, el valor por omisión, todos los registros se transfieren en la primera operación, lo cual solamente es aconsejable para tablas pequeñas.

A partir de ahí, podemos crear y configurar los objetos de acceso a campos igual que si estuviésemos tratando con una tabla o una consulta. Si me hizo caso e incluyó la opción *poIncFieldProps* en el proveedor, se podrá ahorrar la configuración de los campos. He aquí el módulo de datos local después de colocar todos los componentes necesarios:



Ya podemos traer una rejilla, o los componentes visuales que deseemos, al formulario principal y enlazar a éste con el módulo local, para visualizar los datos extraídos del servidor de aplicaciones.

Grabación de datos

Si realizamos modificaciones en los datos por medio de la aplicación cliente, salimos de la misma y volvemos a ejecutarla, nos encontraremos que hemos perdido las actualizaciones. Las aplicaciones que utilizan *TClientDataSet* son parecidas a las que aprovechan las actualizaciones en caché: tenemos que efectuar una operación explícita de actualización del servidor para que éste reconozca los cambios producidos en los datos.

Ya hemos visto que *TClientDataSet* guarda los cambios realizados desde que se cargan los datos en la propiedad *Delta*, y que *ChangeCount* almacena el número de cambios. En las aplicaciones basadas en ficheros planos, los cambios se aplicaban mediante el método *MergeChangeLog*. Ahora, utilizado proveedores como origen de datos, la primera regla del juego dice:

"Prohibido utilizar MergeChangeLog con proveedores"

Para grabar los cambios en el proveedor, debemos enviar el contenido de *Delta* utilizando la misma interfaz *IProvider* con la que rellenamos *Data*. El siguiente método es la forma más fácil de enviar las actualizaciones al servidor:

```
int __fastcall TClientDataSet::ApplyUpdates(int MaxErrors);
```

El parámetro *MaxErrors* representa el número máximo de errores tolerables, antes de abortar la operación:

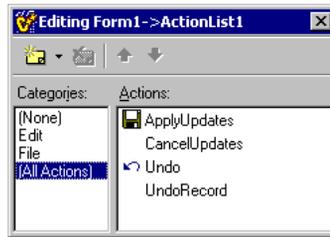
- Si indicamos -1, la operación tolera cualquier número de errores.
- Si indicamos 0, se detiene al producirse el primer error.
- Si indicamos $n > 0$, se admite ese número de errores antes de abortar todo el proceso.

Estamos hablando, por supuesto, de C++ Builder 4. En la versión 3 de la VCL, el valor -1 indicaba que no se toleraban errores. Otra diferencia importante es que el servidor en C++ Builder 4 inicia una transacción sobre la base de datos si no hemos iniciado una explícitamente. Por lo tanto, cuando hablamos de abortar la operación, en realidad estamos hablando de realizar un *Rollback* sobre la base de datos en el servidor. Más adelante veremos con detalle qué sucede cuando se producen errores, y cómo podemos controlar estos errores, tanto en el cliente como en el servidor Mi-

das. Además, veremos cómo manejar transacciones más complejas que las simples transacciones automáticas de los proveedores Midas.

¿Cuándo debemos llamar al método *ApplyUpdates*? Depende del modo de edición y de los convenios de interfaz que usted establezca con sus usuarios. Si el usuario utiliza cuadros de diálogo para insertar o modificar registros, es relativamente sencillo asociar la grabación al cierre del diálogo. Las cosas se complican cuando las actualizaciones se realizan directamente sobre una rejilla. Personalmente, soy partidario en tales casos de que el usuario considere a la aplicación como una especie de procesador de textos, y que disponga de un comando *Guardar* para enviar los cambios al servidor. Realmente, ésta es la metáfora más apropiada para los conjuntos de datos clientes, que no mantienen una conexión directa con los datos originales.

Voy a mostrar cómo incluir el comando *Guardar* y los demás comandos de edición, como *Desbacer*, *Desbacer todo* y *Desbacer registro*, en una lista de acciones de C++ Builder 4. Colocamos un objeto de tipo *TActionList* (página *Standard*) en el módulo de datos, para que esté “cerca” del conjunto de datos cliente. Podemos traer también una lista de imágenes para asociarle jeroglíficos a las acciones. Estas son las acciones que debemos crear:



El evento *OnUpdate* de las acciones individuales se dispara para que podamos activar o desactivar la acción en dependencia del estado de la aplicación. Las tres primeras acciones de nuestro ejemplo se activan cuando hemos realizado cambios sobre el conjunto de datos clientes, por lo que utilizaremos una respuesta compartida para el evento *OnUpdate* de todas ellas:

```
void __fastcall TmodDatos::HayCambios(TObject *Sender)
{
    static_cast<TAction*>(Sender)->Enabled =
        ClientDataSet1->ChangeCount > 0;
}
```

La acción *UndoRecord*, por el contrario, solamente es aplicable cuando el registro activo del conjunto de datos tiene modificaciones, algo que podemos comprobar con la ya conocida función *UpdateStatus*:

```

void __fastcall TmodDatos::RegistroModificado(TObject *Sender)
{
    UndoRecord->Enabled =
        ClientDataSet1->UpdateStatus() != usUnmodified;
}

```

La respuesta de cada acción se programa en el evento *OnExecute*:

```

void __fastcall TmodDatos::ApplyUpdatesExecute(TObject *Sender)
{
    ClientDataSet1->ApplyUpdates(0);
}

void __fastcall TmodDatos::CancelUpdatesExecute(TObject *Sender)
{
    ClientDataSet1->CancelUpdates();
}

void __fastcall TmodDatos::UndoExecute(TObject *Sender)
{
    ClientDataSet1->UndoLastChange(True);
}

void __fastcall TmodDatos::UndoRecordExecute(TObject *Sender)
{
    ClientDataSet1->RevertRecord();
}

```

Estas acciones que hemos preparado son objetos no visuales. Para materializarlas creamos un menú, y en vez de configurar sus comandos laboriosamente, propiedad por propiedad, podemos asignar acciones en la propiedad *Action* de cada *TMenuItem*. La misma operación puede efectuarse sobre los botones *TToolButton* de una barra de herramientas.

Es conveniente, para terminar, programar la respuesta del evento *OnCloseQuery*, de forma similar a como lo haríamos en un procesador de textos:

```

void __fastcall TwndMain::FormCloseQuery(TObject *Sender,
    bool &CanClose)
{
    if (modDatos->ClientDataSet1->ChangeCount > 0)
        switch (MessageBox(0, "¿Desea guardar los cambios?",
            "Confirmar", MB_YESNOCANCEL))
        {
            case IDCANCEL:
                CanClose = False;
                break;
            case IDYES:
                modDatos->ClientDataSet1->ApplyUpdates(0);
                CanClose = modDatos->ClientDataSet1->ChangeCount == 0;
                break;
        }
}

```

Resolución

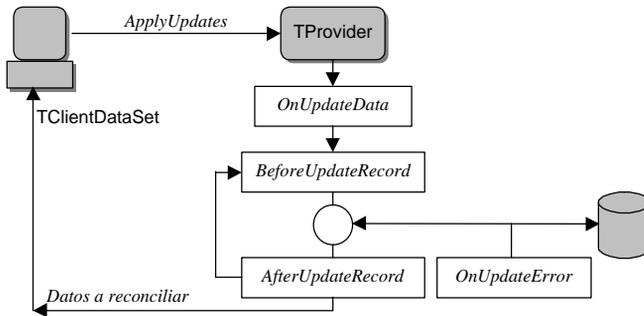
El algoritmo empleado para grabar los cambios efectuados en el cliente mediante el servidor remoto es un ballet muy bien sincronizado y ensayado, cuyas *primas ballerinas* son el cliente y el proveedor, pero en el que interviene todo un coro de componentes. La obra comienza cuando el cliente pide la grabación de sus cambios mediante el método *ApplyUpdates*:

```
ClientDataSet1->ApplyUpdates(0);
```

Internamente, el conjunto de datos echa mano de su interfaz *Provider*, ejecuta el siguiente método remoto, y se queda a la espera del resultado para efectuar una operación que estudiaremos más adelante, llamada *reconciliación*:

```
Provider->ApplyUpdates(Delta, MaxErrors, Result);
```

Como vemos, al servidor *Midas* se le envía *Delta*: el vector que contiene los cambios diferenciales con respecto a los datos originales. Y ahora cambia el escenario de la danza, pasando la acción al servidor:



El proveedor utiliza un pequeño componente auxiliar, derivado de la clase *TCustomResolver*, que se encarga del proceso de grabar las diferencias en la base de datos. En la jerga de Borland/Inprise, este proceso recibe el nombre de *resolución*. El algoritmo de resolución puede concretarse de muchas formas diferentes. El componente *TProvider*, por ejemplo, aplica las modificaciones mediante sentencias SQL que lanza directamente a la base de datos, saltándose el conjunto de datos que tiene conectado en su propiedad *DataSet*. Así se logra mayor eficiencia, pero se asume que *DataSet* apunta a un conjunto de datos del BDE. El componente *TDataSetProvider*, que también se encuentra en la página *Midas*, aplica las actualizaciones utilizando como intermediario a su conjunto de datos asociado. Es más lento, pero puede utilizarse con cualquier tipo de conjunto de datos. Y si usted tiene la suficiente paciencia, puede crearse su proveedor personalizado, derivando clases a partir de *TBaseProvider* y de *TCustomResolver*.

Veamos la secuencia de eventos que se disparan durante la resolución. El primer evento se activa una sola vez, antes de que comience la acción. Se trata de *OnUpdateData*, y puede utilizarse para editar los datos antes de que sean grabados. Su tipo es el siguiente:

```
typedef void __fastcall (__closure *TProviderDataEvent)
(TObject *Sender, TClientDataSet *DataSet);
```

El parámetro *DataSet* contiene un conjunto de datos cliente creado al vuelo por el proveedor, que contiene la información pasada en *Delta*. Podemos recorrer este conjunto de datos, a diferencia de lo que sucede en otros eventos que prohíben mover la fila activa. ¿Qué podemos hacer aquí? Tenemos la oportunidad de modificar registros o completar información. El ejemplo que viene en el Manual del Desarrollador muestra cómo llenar un campo con el momento en que se produce la grabación, para los registros nuevos:

```
void __fastcall TRemMod::Provider1UpdateData(TObject *Sender,
TClientDataSet *DataSet)
{
    DataSet->First();
    while (! DataSet->Eof)
    {
        if (DataSet->UpdateStatus() == usInserted)
        {
            DataSet->Edit();
            DataSet->FieldValues["FechaAlta"] = Now();
            DataSet->Post();
        }
        DataSet->Next();
    }
}
```

Pero también se puede utilizar el evento para controlar el contenido de las sentencias SQL que va a lanzar el proveedor al servidor de datos. Antes de llegar a este extremo, permítame que mencione la propiedad *UpdateMode* del proveedor, que funciona de modo similar a la propiedad homónima de las tablas. Con *UpdateMode* controlamos qué campos deben aparecer en la cláusula **where** de una modificación (**update**) y de un borrado (**delete**). Como expliqué en el capítulo 26, nos podemos ahorrar muchos conflictos asignando *upWhereChanged* a la propiedad, de forma tal que solamente aparezcan en la cláusula **where** las columnas de la clave primaria y aquellas que han sufrido cambios.

¿Necesita más control sobre las instrucciones SQL generadas para la resolución? Debe entonces configurar, campo por campo, la propiedad *ProviderFlags* de los mismos, que ha sido introducida en C++ Builder 4:

Opción	Descripción
<i>pfInWhere</i>	El campo no aparece en la cláusula where
<i>pfInUpdate</i>	El campo no aparece en la cláusula set de las modificaciones

Opción	Descripción
<i>pfInKey</i>	Utilizado para releer el registro
<i>pfHidden</i>	Impide que el cliente vea este campo

Analizando fríamente la lista de opciones anterior, vemos que a pesar de la mayor complejidad, no hay muchas más posibilidades reales que las básicas de *UpdateMode*. Además, los nombres de las dos primeras opciones inducen a pensar completamente en lo contrario de lo que hacen. Cosas de la vida.

Después de haber sobrevivido a *OnUpdateData*, el componente de resolución aplica los cambios en un bucle que recorre cada registro. Antes de cada grabación se activa el evento *BeforeUpdateRecord*, se intenta la grabación y se llama posteriormente al evento *AfterUpdateRecord*. *BeforeUpdateRecord* sirve para los mismos propósitos que *OnUpdateData*, pero esta vez los cambios que realizamos afectarán sólo al registro activo. Este es su prototipo:

```
typedef void __fastcall (__closure *TBeforeUpdateRecordEvent)
(TObject *Sender, TDataSet *SourceDS, TClientDataSet *DeltaDS,
 TUpdateKind UpdateKind, bool &Applied);
```

Nuevamente, podemos utilizar este evento para modificar la forma en que se aplicará la actualización. El registro activo de *DeltaDS* es el que contiene los cambios. Por ejemplo, éste es un buen momento para asignar claves únicas de forma automática. Pero también podemos optar por grabar nosotros mismos el registro, o eliminarlo. Muchas bases de datos están diseñadas con procedimientos almacenados como mecanismo de actualización. Si durante la respuesta a *BeforeUpdateRecord* ejecutamos uno de esos procedimientos almacenados y asignamos *True* al parámetro *Applied*, el proveedor considerará que el registro ya ha sido modificado, y pasará por alto el registro activo.

Finalmente, después de la grabación exitosa del registro, se dispara el evento *AfterUpdateEvent*, que es similar al anterior, pero sin el parámetro *Applied*:

```
typedef void __fastcall (__closure *TAfterUpdateRecordEvent)
(TObject *Sender, TDataSet *SourceDS, TClientDataSet *DeltaDS,
 TUpdateKind UpdateKind);
```

Este evento es de poco interés para nosotros.

Control de errores durante la resolución

Cada vez que se produce un error durante la resolución se dispara el evento *OnUpdateError*, cuya declaración en C++ Builder 4 es:

```

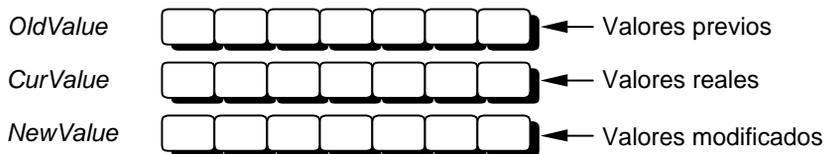
enum TUpdateKind
{ ukModify, ukInsert, ukDelete };

enum TResolverResponse
{ rrSkip, rrAbort, rrMerge, rrApply, rrIgnore };

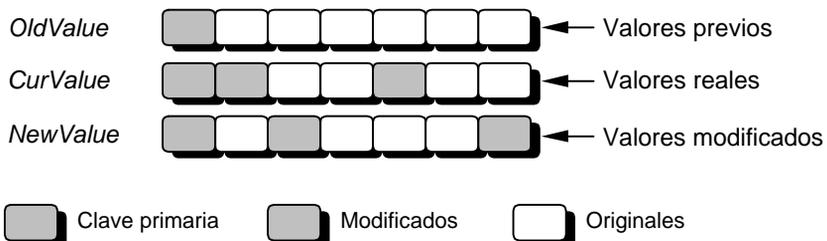
typedef void __fastcall (__closure *TResolverErrorEvent)
(TObject* Sender, TClientDataSet* DataSet,
EUpdateError *E, TUpdateKind UpdateKind,
TResolverResponse &Response);

```

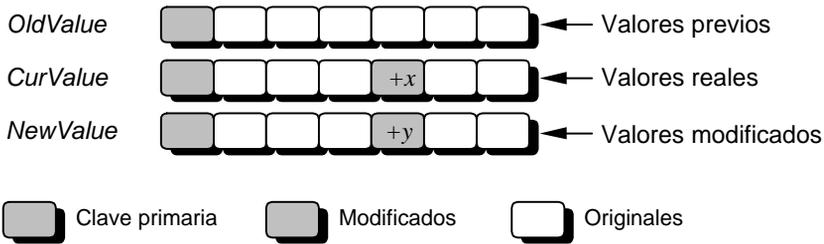
En términos generales, se recomienda realizar la recuperación de errores en las aplicaciones clientes, donde el usuario puede intervenir con mayor conocimiento de causa. Sin embargo, existen situaciones especiales en las que los conflictos pueden resolverse automáticamente en el servidor. Por supuesto, debemos en lo posible tratar estos errores durante la resolución. La corrección de errores se hace más fácil, tanto en la resolución como en la posterior reconciliación, gracias a la existencia de una propiedad asociada a los campos, *CurValue*, que indica cuál es el valor del campo encontrado en el registro de la base de datos, si se produce un fallo del bloqueo optimista. El siguiente esquema representa la relación entre las propiedades *OldValue*, *CurValue* y *NewValue*.



Supongamos por un momento que el valor de *UpdateMode* del proveedor sigue siendo *upWhereAll*. Si dos usuarios realizan actualizaciones en diferentes columnas, sin llegar a afectar a la clave primaria, estamos ante una situación conocida como *actualizaciones ortogonales* (*orthogonal updates*), y se produce un error. Cuando el segundo usuario que intenta grabar recibe el correspondiente error, el estado de los campos puede representarse mediante el siguiente diagrama:



Otro caso frecuente se produce cuando dos aplicaciones diferentes intentan modificar la misma columna, y el valor numérico almacenado en la misma representa un acumulado:



Todos estos tipos de conflicto pueden solucionarse automáticamente dentro del evento *OnUpdateRecord* del servidor. Veamos, por ejemplo, cómo puede resolverse un conflicto con una actualización acumulativa sobre el salario de un empleado. Comenzaré definiendo una función auxiliar que compruebe que solamente se ha cambiado el salario en una actualización:

```

bool __fastcall SalaryChanged(TDataSet *ADataset)
{
    for (int i = ADataset->FieldCount - 1; i >= 0; i--)
    {
        TField *f = ADataset->Fields->Fields[i];
        bool Changed = ! VarIsEmpty(f->CurValue)
            && ! VarIsEmpty(f->NewValue);
        if (Changed != (CompareText(f->FieldName, "SALARY") != 0))
            return False;
    }
    return True;
}

```

La función se basa en que *CurValue* y *NewValue* valen *Unassigned* cuando el valor que representan no ha sido modificado. Recuerde que *VarIsEmpty* es la forma más segura de saber si un *Variant* contiene este valor especial. La regla anterior tiene una curiosa excepción: si *CurValue* y *NewValue* han sido modificados con el mismo valor, la propiedad *CurValue* también contiene *Unassigned*:

```

void __fastcall TMastSql::EmpleadosUpdateError(TObject *Sender,
    TClientDataSet *DataSet, EUpdateError *E,
    TUpdateKind UpdateKind, TResolverResponse &Response)
{
    if (UpdateKind == ukModify && SalaryChanged(DataSet))
    {
        TField *f = DataSet->FieldByName("SALARY");
        Currency CurSal = VarIsEmpty(f->CurValue) ?
            f->NewValue : f->CurValue;
        f->NewValue = CurSal + f->NewValue - f->OldValue;
        Response = rrApply;
    }
}

```

Reconciliación

Como todas las modificaciones a los datos tienen lugar en la memoria de la estación de trabajo, la grabación de las modificaciones en el servidor presenta los mismos problemas que las actualizaciones en caché, como consecuencia del comportamiento optimista. El mecanismo de resolución de conflictos, denominado *reconciliación*, es similar al ofrecido por las actualizaciones en caché, y consiste en un evento de *TClientDataSet*, cuyo nombre es *OnReconcileError* y su tipo es el siguiente:

```
enum TUpdateKind
{ ukModify, ukInsert, ukDelete };

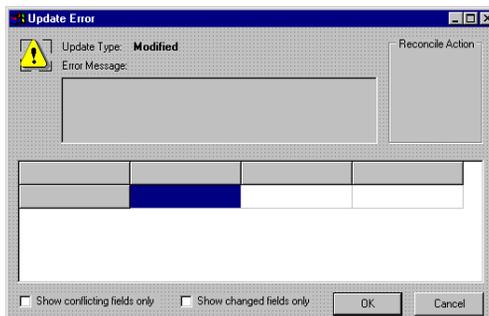
enum TReconcileAction
{ raSkip, raAbort, raMerge, raCorrect, raCancel, raRefresh };

typedef void __fastcall (__closure *TReconcileErrorEvent)
(TClientDataSet *DataSet, EReconcileError *E,
TUpdateKind UpdateKind, TReconcileAction &Action);
```

En contraste con las actualizaciones en caché, hay más acciones posibles durante la reconciliación de los datos en este modelo. Los valores del tipo *TReconcile.Action* son:

Valor	Significado
<i>raSkip</i>	El registro no se actualiza, pero permanece en la lista de cambios.
<i>raAbort</i>	Se aborta la operación de reconciliación.
<i>raMerge</i>	Se mezclan los cambios con los del registro actual.
<i>raCorrect</i>	Se reintenta la grabación.
<i>raCancel</i>	No se actualiza el registro, definitivamente.
<i>raRefresh</i>	Se relea el registro.

La forma más conveniente de reconciliar datos es presentando un cuadro de diálogo de reconciliación, definido como plantilla en la página *Dialogs* del Depósito de Objetos. Para incluirlo en nuestra aplicación, vaya a dicha página del Depósito y seleccione el icono *Reconcile error dialog*. Al proyecto se le añade entonces un formulario de diálogo, con el nombre *TReconcileErrorForm*.



Nada es perfecto: hay que traducir los mensajes del diálogo. Es importante además utilizar el menú *Project|Options* para eliminar este formulario de la lista de ventanas con creación automática. Para garantizar que realizamos la operación, la variable que normalmente aparece en la interfaz, y que se utiliza en la creación automática, ha sido eliminada de la plantilla. La creación y ejecución del diálogo es tarea de la siguiente función, declarada en la cabecera de la unidad asociada:

```
TReconcileAction HandleReconcileError(
    TComponent* Owner, TDataSet *DataSet,
    TUpdateKind UpdateKind, EReconcileError *ReconcileError);
```

El uso típico de la misma durante el evento *OnReconcileError* es como sigue:

```
void __fastcall TDataModule2::tbEmpleadosReconcileError(
    TClientDataSet *DataSet, EReconcileError *E,
    TUpdateKind UpdateKind, TReconcileAction &Action)
{
    Action = HandleReconcileError(NULL, DataSet, UpdateKind, E);
}
```

Si el programador no quiere complicarse demasiado la vida, o no confía en la sensatez de sus usuarios, puede programar un evento de reconciliación más elemental:

```
void __fastcall TDataModule2::tbEmpleadosReconcileError(
    TClientDataSet *DataSet, EReconcileError *E,
    TUpdateKind UpdateKind, TReconcileAction &Action)
{
    ShowMessage("Hay errores");
    Action = raAbort;
}
```

En tal caso es conveniente que los cambios efectuados queden señalados en la rejilla de datos (si es que está utilizando alguna):

```
void __fastcall TForm1::DBGrid1DrawColumnCell(TObject *Sender,
    const TRect &Rect, int DataCol, TColumn *Column,
    TGridDrawState State)
{
    if (DataModule2->ClientDataSet1->UpdateStatus() != usUnmodified)
        DBGrid1->Canvas->Font->Style = TFontStyles() << fsBold;
    DBGrid1->DefaultDrawColumnCell(Rect, DataCol, Column, State);
}
```

Si va a utilizar la función *UpdateStatus* de los registros del conjunto de datos clientes, es aconsejable que llame a *Refresh* después de grabar los cambios exitosamente, pues la grabación no modifica el valor de esta propiedad.

Relaciones *master/detail* y tablas anidadas

Hay dos formas de configurar relaciones *master/detail* en un cliente Midas. Podemos exportar dos interfaces *IProvider* desde el servidor, correspondientes a dos tablas independientes, enganchar a las mismas dos componentes *TClientDataSet* en el módulo cliente, y establecer entonces la relación entre ellos. Haga esto si quiere sabotear la aplicación.

La mejor forma de establecer la relación es configurarla en las tablas del servidor Midas, en el módulo remoto, y exportar solamente el proveedor de la tabla maestra. A partir de la versión 4 de la VCL, el proveedor puede enviar las filas de detalles junto a las filas maestras (la opción más simple y sensata), o esperar a que el cliente pida explícitamente esos datos, mediante la opción *poFetchDetailsOnDemand* de la propiedad *Options*. Cuando programe el cliente, utilice dos *TClientDataSet*. El primero debe conectarse al proveedor de la tabla maestra. Si trae todos los objetos de acceso a campos del componente, verá que se crea también un campo de tipo *TDataSetField*, con el nombre de la tabla de detalles. Este campo se asigna a la propiedad *DataSetField* del segundo conjunto de datos. Y ya tenemos una relación *master/detail* entre los dos componentes.

Hay dos importantes ventajas al utilizar este enfoque para tablas dependientes. La primera: menos tráfico de red, pues la relación se establece en el servidor, y los datos viajan sólo por demanda. La segunda: todas las modificaciones deben acumularse en el *log* del conjunto maestro y deben poder aplicarse en una sola transacción cuando se apliquen las actualizaciones (*ApplyUpdates*) de este conjunto.

Envío de parámetros

Cuando aplicamos un filtro en un *TClientDataSet* la expresión se evalúa en el lado cliente de la aplicación, por lo cual no se limita el conjunto de registros que se transfieren desde el servidor. Existe, sin embargo, la posibilidad de aplicar restricciones en el servidor Midas si utilizamos la propiedad *Params* del conjunto de datos clientes. La estructura de esta propiedad es similar a la propiedad del mismo nombre del componente *TQuery*, pero su funcionamiento es diferente, ya que requiere coordinación con el servidor de aplicaciones.

Si el proveedor al cual se conecta el conjunto de datos clientes está exportando una consulta desde el servidor, se asume que los parámetros del *TClientDataSet* coinciden en nombre y tipo con los de la consulta. Cuando cambiamos el valor de un parámetro en el cliente, el valor se transmite al servidor y se evalúa la consulta con el nuevo parámetro. Hay dos formas de asignar un parámetro en un conjunto de datos cliente. La primera es cerrar el conjunto de datos, asignar valores a los parámetros deseados y volver a abrirlo:

```
ClientDataSet1->Close();  
ClientDataSet1->Params->ParamByName("CIUDAD")->AsString =  
    "Cantalapiedra";  
ClientDataSet1->Open();
```

La otra es asignar el parámetro sin cerrar el conjunto de datos, y aplicar el método *SendParams*:

```
ClientDataSet1->Params->ParamByName("CIUDAD")->AsString =  
    "Cantalapiedra";  
ClientDataSet1->SendParams();
```

El método *FetchParams* realiza la acción contraria a *SendParams*: lee el valor actual de un parámetro desde el servidor de aplicaciones.

¿Qué sucede cuando el conjunto de datos que se exporta desde el servidor es una tabla? La documentación de C++ Builder establece que, en ese caso, el nombre de los parámetros del *TClientDataSet* debe corresponder a nombres de columnas de la tabla, y que la tabla restringe el conjunto de registros activos como si se tratara de un filtro sobre dichos campos. Sin embargo, este humilde programador ha examinado el código fuente (procedimiento *SetParams* del componente *TProvider*), y ha encontrado que la VCL intenta establecer un rango en el servidor. Hay entonces una diferencia clave entre lo que dice la documentación y lo que sucede en la práctica, pues el rango necesita que los campos afectados sirvan de criterio de ordenación activo. Tampoco he logrado hacer que funcione *SendParams* con el conjunto de datos clientes abierto: es preferible cerrar el conjunto de datos, asignar parámetros y volver a abrirlo.

Extendiendo la interfaz del servidor

En la versión 3 de la VCL, la resolución de las modificaciones efectuadas en el cliente no transcurría, por omisión, dentro de una transacción que garantizara su atomicidad. Esto era una fuente de problemas cuando editábamos objetos complejos, distribuidos entre varias tablas, como sucede al trabajar con una relación *master/detail*. Para resolver esta dificultad, no quedaba más remedio que definir métodos dentro del objeto de automatización y exportarlos, de modo que las actualizaciones en el cliente hicieran uso de los mismos.

Como he dicho antes, la versión 4 sí utiliza transacciones automáticamente, cuando no las hemos iniciado de forma explícita. Ahora bien, ¿cómo puede una aplicación cliente remota iniciar una transacción mediante objetos situados en el servidor? La respuesta es simple: el módulo de datos remoto implementa, en definitiva, una interfaz de automatización. Si el servidor exporta otros métodos de automatización que los predefinidos, el cliente puede ejecutarlos a través de la siguiente propiedad de la conexión al servidor:

```
__property Variant AppServer;
```

De modo que regresamos al servidor de aplicaciones y abrimos la biblioteca de tipos, con el comando de menú *View | Type library*. En el nodo correspondiente a la interfaz *IMastSql* añadimos tres métodos, de los cuales muestro su declaración en IDL:

```
HRESULT _stdcall StartTransaction( void );
HRESULT _stdcall Commit( void );
HRESULT _stdcall Rollback( void );
```

En la unidad de implementación del módulo remoto proporcionamos el cuerpo a los métodos anteriores:

```
STDMETHODIMP TMastSQLImpl::StartTransaction()
{
    try
    {
        TDatabase *db = m_DataModule->tbClientes->Database;
        if (! db->IsSQLBased) db->TransIsolation = tiDirtyRead;
        db->StartTransaction();
    }
    catch(Exception &e)
    {
        return Error(e.Message.c_str(), IID_IMastSQL);
    }
    return S_OK;
}

STDMETHODIMP TMastSQLImpl::Commit()
{
    try
    {
        m_DataModule->tbClientes->Database->Commit();
    }
    catch(Exception &e)
    {
        return Error(e.Message.c_str(), IID_IMastSQL);
    }
    return S_OK;
}

STDMETHODIMP TMastSQLImpl::Rollback()
{
    try
    {
        m_DataModule->tbClientes->Database->Rollback();
    }
    catch(Exception &e)
    {
        return Error(e.Message.c_str(), IID_IMastSQL);
    }
    return S_OK;
}
```

El truco más importante consiste en cómo acceder al módulo de datos (el objeto de clase *TMastSQL*) desde el objeto de automatización, de clase *TMastSQLImpl*: hay que

utilizar el atributo *m_DataModule* de esta última clase. Observe también cómo protegemos al servidor del lanzamiento de excepciones, propagando éstas al cliente por medio del código de retorno de los métodos exportados.

En Delphi no es necesario arropar los métodos de automatización de un servidor dentro de instrucciones de captura de excepciones, pues la directiva **safecall** genera por nosotros el código necesario.

Ahora desde el cliente podemos ejecutar secuencias de instrucciones como la siguiente:

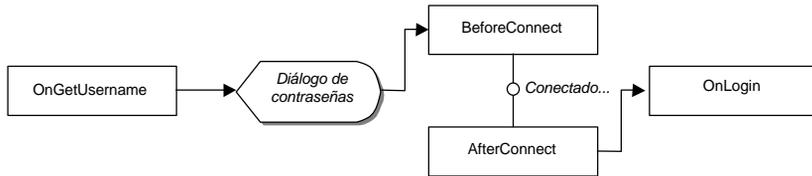
```
// ...
DCOMConnection1->AppServer.Exec(Procedure("StartTransaction"));
try
{
    // Aplicar cambios en varios conjuntos de datos
    DCOMConnection1->AppServer.Exec(Procedure("Commit"));
}
catch(Exception&)
{
    DCOMConnection1->AppServer.Exec(Procedure("Rollback"));
    throw;
}
// ...
```

La técnica anterior tiene muchas aplicaciones, como poder activar procedimientos almacenados situados en el módulo remoto desde las aplicaciones clientes. Más adelante veremos cómo y cuándo aprovechar las interfaces duales para acelerar las llamadas al servidor.

Alguien llama a mi puerta

En la mayoría de los casos, la capa intermedia de una aplicación desarrollada con Midas accederá a una bases de datos SQL, que probablemente controlará su uso mediante contraseñas. ¿Cómo puede un usuario remoto suministrar su nombre y contraseña a un servidor Midas para que éste, a su vez, realice la conexión a la base de datos? ¿Hay algún tipo de magia en juego? No, no hay magia, pero sí una propiedad y un evento.

La propiedad se llama *LoginPrompt*, y se define en la clase *TDispatchConnection*, por lo cual está disponible en *TDCOMConnection*, el componente que hemos estado empleando para conectarnos al servidor, pero también podremos utilizarla en *TSocketConnection* y *TOLEnterpriseConnection*, que estudiaremos más adelante. Cuando *LoginPrompt* está activa, *TDCOMConnection* produce la siguiente secuencia de eventos durante su conexión al módulo remoto:



El primer evento que se dispara, *OnGetUsername*, nos da la posibilidad de inicializar el nombre de usuario que se muestra en el diálogo de petición contraseñas. Podemos, por ejemplo, utilizar el nombre con el que el sistema operativo conoce al usuario:

```

void __fastcall TDataModule1::DCOMConnection1GetUsername(
    TObject *Sender, AnsiString &Username)
{
    char buffer[128];
    DWORD len = sizeof(buffer);
    Win32Check(GetUserName(buffer, &len));
    Username = buffer;
}
  
```

Entonces aparece en pantalla el diálogo de identificación, para que el usuario teclee su contraseña y corrija su nombre, si es necesario. Si se cancela el diálogo la conexión se aborta, naturalmente. En caso contrario, se dispara el evento *BeforeConnect*, se establece la conexión en sí, se activa *AfterConnect ...* y el control de la ejecución vuelve a nuestras manos, pues tenemos que dar respuesta al evento *OnLogin*, también de la clase *TDCOMConnection*:

```

void __fastcall TDataModule1::DCOMConnection1Login(
    TObject *Sender, AnsiString Username, AnsiString Password)
{
    DCOMConnection1->AppServer.Exec(
        Procedure("Identificar") << Username << Password);
}
  
```

Aquí termina todo lo que Midas hace para ayudarnos: sencillamente nos dice quién se ha conectado, y con cuál contraseña. Es responsabilidad nuestra comunicárselo al servidor, y lo hemos hecho llamando a un método diseñado por nosotros e implementado por el servidor, de nombre *Identificar*, y que recibe como parámetros el nombre del usuario y su contraseña. El método puede asignar los parámetros *USER NAME* y *PASSWORD* en la propiedad *Params* de un *TDatabase*, para activar este componente posteriormente.

La metáfora del maletín

Paco McFarland, empleado de la mítica Anaconda Software, inicia todos los lunes por la mañana desde casa, al terminar de cepillarse los dientes, una sesión de acceso remoto telefónico entre su portátil y el servidor de aplicaciones de la compañía. Una vez establecida la conexión, ejecuta una aplicación que accede a los datos de los clientes de Anaconda y los guarda en el disco duro del portátil; a continuación, y aunque la línea la paga la empresa, corta el cordón umbilical de la conexión. El trabajo de Paco consiste en realizar visitas programadas a determinados clientes y, como resultado de las mismas, actualizar los datos correspondientes en el portátil. Al terminar el día, y antes de ingerir sus vitaminas y analgésicos, Paco vuelve a enchufar el portátil al servidor de aplicaciones e intenta enviar los registros modificados al ordenador de la empresa. Por supuesto, es posible que encuentre conflictos en determinados registros, pero para esto cuenta con técnicas de reconciliación apropiadas.

La historia anterior muestra en acción el “modelo del maletín” (*briefcase model*). Este modelo puede implementarse en C++ Builder utilizando el mismo componente *TClientDataSet*. Para soportar el modelo, el componente ofrece los métodos *LoadFromFile* y *SaveToFile*; estas rutinas trabajan con ficheros “planos” en formato ASCII y ya las hemos estudiado en el capítulo 32. Una aplicación cliente puede incluir opciones para guardar sus datos en un fichero de este tipo y para recuperar los datos posteriormente, sobre todo en el caso en que no se detecta un servidor de aplicaciones al alcance del ordenador.

Tipos de conexión

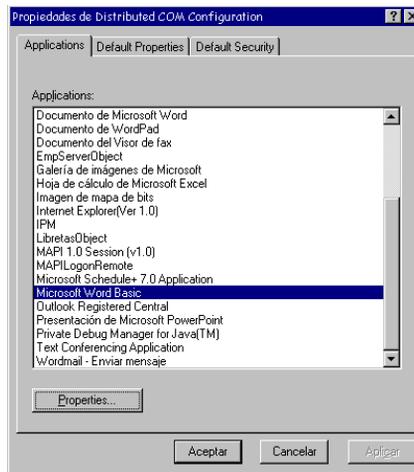
Todos los ejemplos que he mostrado utilizan el componente *TDCOMConnection* para establecer la comunicación con el servidor Midas. He utilizado dicho componente como punto de partida pensando en el lector que realiza sus pruebas cómodamente en casa, en su ordenador personal, y que al no disponer de una red para probar técnicas sofisticadas de comunicación, sitúa al cliente Midas y a su servidor en el único nodo disponible. Veamos ahora los distintos tipos de conexiones que pueden establecerse y los problemas de configuración de cada uno de ellos.

El primer tipo de conexión se basa en COM/DCOM. Ya hemos visto que cuando el cliente y el servidor residen en la misma máquina es éste el tipo de protocolo que utilizan para comunicarse, y no hace falta ninguna configuración especial. Así que vamos directamente a DCOM. Para poder utilizar este protocolo, debe tener los siguientes elementos:

- Una red Windows/Windows NT en la que los ordenadores controlen el acceso a los recursos compartidos mediante el nombre de usuarios.

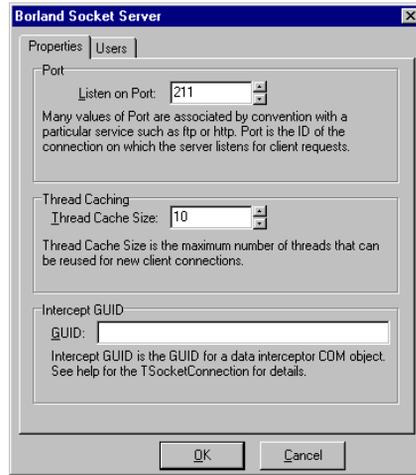
- Si las estaciones de trabajo tienen instalado Windows 95/98, el requisito anterior obliga a que estén conectadas a un dominio NT. Así que, en cualquier caso, necesitará un ordenador que actúe como controlador de dominio de Windows NT.
- Los ordenadores que tienen Windows 95 necesitan un parche de Microsoft para activar DCOM, tanto si van a actuar como clientes o como servidores. Si se trata de Windows 98, NT Server o Workstation, el soporte DCOM viene incorporado.

Una de las rarezas de DCOM es que no se configura, como sería de esperar, mediante el Panel de Control. Hay que ejecutar, desde la línea de comandos o desde *Inicio | Ejecutar*, el programa *dcomcnfg.exe*. La versión para Windows 95 permite activar o desactivar DCOM en el ordenador local, principalmente:



La versión de *dcomcnfg* para Windows NT se utiliza para controlar la seguridad del acceso a las aplicaciones situadas en el servidor. El control de acceso es la principal ventaja que ofrece DCOM respecto a otros protocolos. Sin embargo, a veces es un poco complicado echar a andar DCOM en una red existente, configurada de acuerdo a las prácticas viciosas y perversas de algunos “administradores de redes”.

En tales casos, la forma más sencilla de comunicación es utilizar el propio protocolo TCP/IP, instalando en el ordenador que contiene el servidor Midas una aplicación que actúa como *proxy*, o delegada, y que traduce las peticiones TCP/IP en llamadas a métodos COM dentro del servidor. Esta aplicación se llama *socksrrr.exe*, y se encuentra en el directorio *bin* de C++ Builder:



La aplicación al ejecutarse se coloca en la bandeja de iconos; la imagen anterior corresponde en realidad al diálogo de propiedades de la misma. También hay una versión, *socksrv.exe*, que se ejecuta como un servicio en Windows NT. Si queremos utilizar TCP/IP con Midas, debemos ejecutar cualquiera de estas dos versiones antes de conectar el primer cliente al servidor. Da lo mismo que la máquina que contiene el servidor ejecute Windows 95, 98 o NT. ¿Qué modificaciones debemos realizar en las aplicaciones clientes y servidoras para que se conecten vía TCP/IP? En el servidor, ninguna. En el cliente debemos sustituir el componente *TDCOMConnection* por un *TSocketConnection*. El uso de este tipo de conexión puede disminuir el tráfico en la red. DCOM envía periódicamente desde el servidor a su cliente notificaciones, que le permiten saber si sus clientes siguen activos y no se ha producido una desconexión. Así pueden ahorrarse recursos en caso de fallos, pero las notificaciones deben viajar por la red. *TSocketConnection* evita este tráfico, pero un servidor puede perder a un cliente y no darse cuenta. Este tipo de conexión tampoco soporta el control de acceso basado en roles de DCOM.

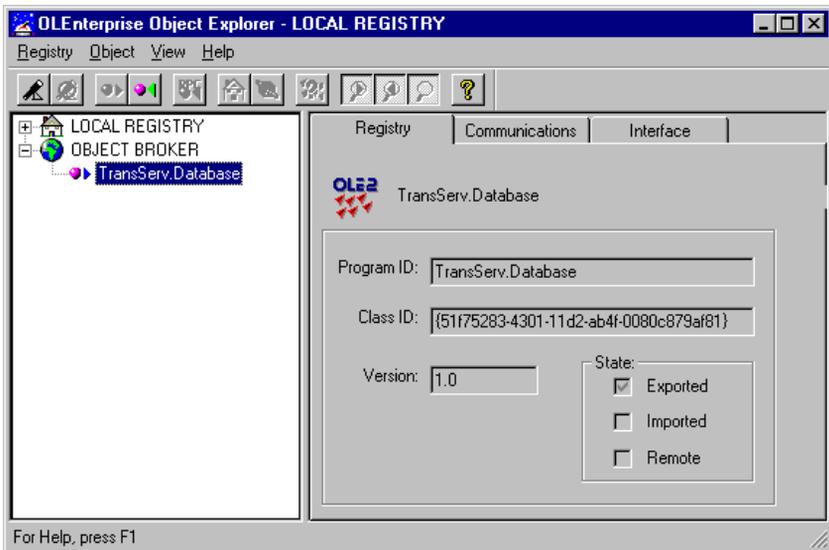
OLEEnterprise es el tercer mecanismo de comunicación. Está basado en RPC, funciona en cualquiera de las variantes de Windows, y permite implementar de forma fácil estas tres deseables características:

- **Transparencia de la ubicación:** El cliente no tiene por qué conocer el ordenador exacto donde está situado su servidor. Todo lo que tiene que especificar es el nombre del servicio (*ServerGUID*) y el nombre del ordenador donde se ejecuta el Agente de Objetos (*Object Broker*). Este último es el software central de OLEEnterprise, que lleva un directorio de servicios para toda la red.
- **Balance de carga:** Gracias a su arquitectura, el Agente de Objetos puede decidir a qué servidor se conecta cada cliente. Al principio de este capítulo mencionábamos la posibilidad de habilitar una batería de servidores Midas redundante para

reducir el tráfico en red. Bien, el balance de carga de OLEnterprise es la técnica más completa y segura para aprovechar esta configuración.

- **Seguridad contra fallos:** El uso de una batería de servidores redundantes permite que, cuando se cae un servidor, sus clientes puedan deshacer las transacciones pendientes, si es que existen, y conectarse a otro servidor de la lista.

La gran desventaja de OLEnterprise es que debemos instalarlo tanto en los servidores como en los clientes, y se trata de una instalación bastante pesada. La imagen de la página siguiente corresponde al *Object Explorer*, la herramienta de OLEnterprise que permite registrar para su uso global a las aplicaciones situadas en los servidores de capa intermedia:



Balance de carga simple

Con C++ Builder 4 no hace falta OLEnterprise para disponer de un mecanismo sencillo de balance de la carga de los servidores. La técnica alternativa está basada en el componente *TSimpleObjectBroker*, que es un descendiente de la clase más general *TCustomObjectBroker*. La idea consiste en especificar una lista de servidores, dando sus nombres o sus direcciones IP, dentro de la propiedad *Servers* de un *TSimpleObjectBroker*. Los componentes de conexión, como *TDCOMConnection* y *TSocketConnection*, en vez de especificar el nombre o dirección de un ordenador deben conectarse al agente por medio de la propiedad *ObjectBroker*.

¿Qué sucede cuando se intenta realizar la conexión? El componente de conexión pide un servidor al agente de objetos, y éste devuelve un servidor de la lista de

acuerdo a su propiedad *LoadBalanced*. Si vale *True*, el servidor se elige aleatoriamente. En caso contrario, se selecciona el primer servidor de la lista. ¿Qué sentido tiene entonces utilizar este componente? Bien, en vez de confiar en el azar, usted puede configurar el orden de la lista de servidores en un cliente determinado, una vez instalada la aplicación. Por ejemplo, los ordenadores del departamento comercial se deben conectar siempre a *SVENTAS*, mientras que Investigación y Desarrollo siempre se conectará a *SID*. Por supuesto, esto implica desarrollar código especial para poder realizar esta reconfiguración persistente en tiempo de ejecución.

Interfaces duales en Midas

Si el mecanismo de comunicación que utilizamos es DCOM, podemos utilizar la interfaz dual del módulo remoto en el cliente para acelerar las llamadas a las extensiones exportadas por el módulo, en vez de utilizar la interfaz *IDispatch* que devuelve la propiedad *AppServer* de *TDCOMConnection*. Recuerde que C++ Builder genera un fichero de cabecera, en nuestro ejemplo *Servidor.App_TLB.h*, a partir de la biblioteca de tipos del servidor. Dentro de este fichero se define una clase *IMastSQL* que representa la interfaz del mismo nombre.

Una aplicación cliente puede incluir este fichero, junto a su compañero de extensión *cpp*, para tener acceso a las declaraciones de tipos y constantes. Si nuestro módulo implementaba *IMastSql*, podíamos realizar llamadas a métodos remotos de la siguiente forma:

```

{
    IMastSQL *MastSQL;

    OleCheck(LPDISPATCH(DCOMConnection1->AppServer)->
        QueryInterface(IID_IMastSQL, (void**) &MastSQL));
    try
    {
        OleCheck(MastSQL->StartTransaction());
        try {
            // Aplicar cambios en varios conjuntos de datos
            OleCheck(MastSQL->Commit());
        }
        catch(Exception&) {
            OleCheck(MastSQL->Rollback());
            throw;
        }
    }
    _finally
    { MastSQL->Release(); }
}

```

También podríamos hacer uso de la interfaz “inteligente” *TCOMIMastSQL*, para evitar las llamadas explícitas a los métodos *AddRef* y *Release*.

Si el protocolo no es DCOM, no podemos realizar esta mejora al algoritmo, pues OLEEnterprise y el Servidor de Sockets se limitan a realizar el *marshaling* para la interfaz *IDispatch*. Sin embargo, algo podemos hacer: utilizar la interfaz *IMastSQLDisp*, que también se declara en el fichero obtenido a partir de la biblioteca de tipos:

```
{
    IMastSQLDisp MastSQL;

    MastSQL.Bind(LPDISPATCH(DCOMConnection1->AppServer));
    MastSQL.StartTransaction();
    // ... etc ...
}
```

Como esta interfaz asocia códigos de identificación a los métodos, evitamos el uso de *GetIDOfNames* para recuperar estos códigos, aunque estemos obligados a seguir utilizando implícitamente a *Invoke*. Algo es algo.

Coge el dinero y corre: trabajo sin conexión

Ahora que ya sabemos qué es Midas y cómo funciona, podemos mostrar una de las aplicaciones más frecuentes y útiles de esta técnica. Se trata de la posibilidad de trabajar con copias *off-line* de determinados datos.

- “Espera”, me corrige el lector, “eso es el modelo del maletín”.
- Pues no, porque en ese modelo de programación la copia local de las tablas se almacena en el disco duro del cliente, como ficheros planos, lo cual no sucederá con la nueva técnica.
- “Entonces se trata del trabajo habitual con Midas, que nos ofrece siempre una copia local de los datos para que la consultemos y modifiquemos”.

Tampoco. Cuando nos conectamos a un proveedor remoto, es cierto que vamos leyendo poco a poco conjuntos pequeños de registros de la tabla o consulta original. También es cierto que la edición se realiza sobre la copia local, y que la grabación se produce mediante una operación explícita a iniciativa del usuario. Pero durante todo este proceso la fuente de datos original continúa activa. El cursor abierto en el servidor sigue ocupando recursos de memoria, bloqueos, etc. Como demostraremos, este gasto de recursos puede evitarse mediante la siguiente técnica:

1. El servidor no tiene que exportar la interfaz *IProvider*. Por el contrario, debe exportar al menos un par de métodos: uno para leer datos y otro para recibir las modificaciones. En ambos casos, los datos se transfieren dentro de un parámetro de tipo *OleVariant*, con el formato usual de Midas.
2. Para traer registros al cliente llamamos al primero de los dos métodos mencionados, que *sí* utilizará internamente una interfaz *IProvider* para obtener el paquete de datos correspondiente. El método debe abrir los cursores necesarios, ya sea

- mediante consultas o tablas (preferiblemente consultas en este caso), y cerrarlos una vez que tenga la información solicitada.
3. El paquete de datos se asigna directamente a la propiedad *Data* de un *TClientDataSet*. El usuario navega sobre esta copia y realiza cambios y adiciones sobre la misma. Está claro que, mientras tanto, el servidor se ha podido desentender de nosotros.
 4. Si hay grabar modificaciones, se envía el contenido de la propiedad *Delta* del *TClientDataSet* al segundo método exportado por el servidor. Dentro del mismo, tenemos que llamar manualmente al método *ApplyUpdates* de la interfaz *IProvider*. Si detectamos conflictos en la actualización, debemos enviar los registros con problemas de vuelta al cliente, para su reconciliación.

La condición fundamental para que la técnica explicada funcione es que los conjuntos de datos transferidos deben ser pequeños. Por ejemplo, sería un disparate traerse toda la tabla de clientes a una estación de trabajo. Es muy probable entonces que tengamos que adaptar la interfaz de usuario de nuestra aplicación a estas condiciones de trabajo. Si no podemos realizar la adaptación, entonces debemos olvidarnos de utilizar Midas.

No es necesario que hayamos dividido el sistema en tres capas “físicas” para poder sacar partido de la nueva forma de trabajo. Para simplificar, haremos una demostración con un sistema tradicional en dos capas, en los que la interfaz *IProvider* la proporcionará un *TProvider* ubicado en el mismo módulo de datos que los *TClientDataSet*.

Inicie una aplicación, cree un módulo de datos dentro de la misma y traiga un componente *TDatabase*. Conéctelo al alias *iblocal*, de las demostraciones que trae C++ Builder. Traiga entonces un par de *TQueries*, y asigne las siguientes instrucciones en sus propiedades *SQL* respectivas:

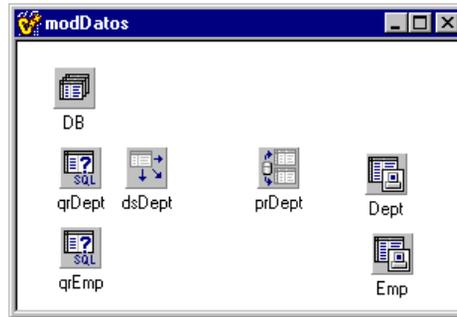
```
qrDept: select *
        from Department
        where Department like :PATTERN
        order by Department asc
```

```
qrEmp: select *
        from Employee
        where Dept_No = :DEPT_NO
```

La primera consulta devuelve aquellos departamentos cuyo nombre es parecido a un patrón que suministraremos. El parámetro *Pattern* debe declararlo mediante el tipo *fiString*. La segunda retorna los empleados pertenecientes al departamento activo de *qrDept*. Hay que traer un *TDataSource* y conectarlo a *qrDept*, para después asignar el nuevo componente en la propiedad *DataSource* de *qrEmp*. De este modo, la consulta

sabe que el parámetro *Dept_No* corresponde a la columna del mismo nombre en la consulta maestra de departamentos.

La siguiente imagen muestra el módulo de datos final, con todos sus componentes. Hasta el momento solamente hemos traído aquellos que acceden directamente al servidor SQL, y los hemos agrupado a la izquierda del módulo:



Debemos entonces traer un componente *TProvider*. Hay que cambiar su *DataSet* para que apunte a *qrDept*, hay que incluir la opción *poIncFieldProps* dentro de *Options*, y es conveniente asignar *npWhereChanged* a la propiedad *UpdateMode*.

Por último, debemos añadir un par de componentes *TClientDataSet*. Para simplificar su configuración, realizamos los siguientes pasos:

- Vaya directamente a la propiedad *ProviderName* del primero de los conjuntos de datos clientes. Asígnele el nombre del proveedor que reside en el módulo.
- Invoque al Editor de Campos del componente y traiga todos los campos asociados desde el proveedor. Observe que hay un último campo cuyo nombre es *qrEmp*, y su tipo *TDataSetField*.
- Seleccione el segundo conjunto de datos y asigne ese campo en la propiedad *DataSetField*. Cree sus objetos de acceso a campo.
- Asegúrese de que las dos consultas y los dos conjuntos de datos clientes estén cerrados en tiempo de diseño.

Para leer y grabar datos de departamentos debemos crear un par de métodos públicos en el módulo de datos. Si se tratase de un servidor Midas remoto, los métodos deberían pertenecer a la interfaz del módulo. Primero tenemos el método de lectura:

```
void __fastcall TmodDatos::Load(const AnsiString DeptName)
{
    int RecsOut;

    qrDept->ParamByName("Pattern")->AsString = DeptName;
    qrDept->Open();
    qrEmp->Open();
}
```

```

Dept->Data = prDept->GetRecords(0, RecsOut);
Dept->AppendData(prDept->GetRecords(1, RecsOut), True);
Emp->Open();
qrEmp->Close();
qrDept->Close();
}

```

Load abre las dos consultas después de asignar el patrón de la búsqueda en la consulta maestra. Como nuestro ejemplo utiliza InterBase como servidor SQL, no tenemos una forma sencilla de limitar el número de filas devuelto por la consulta principal; recuerde que DB2, MS SQL y Oracle tienen extensiones para especificar cuántos registros queremos como máximo. Si dispusiéramos de tales extensiones, podríamos haber sustituido las dos instrucciones que llaman a *GetRecords* por ésta más simple:

```
Dept->Data = prDept->Data;
```

Cuando se pide la propiedad *Data* de un proveedor, éste recupera todas las filas del conjunto de datos asociado. Por este motivo es que llamamos cautelosamente a *GetRecords*. La primera llamada recupera la información de “esquema” de la consulta, mientras que con la segunda leemos realmente el primer registro, si es que existe. El método *Load* termina abriendo el conjunto de datos de detalles, y cerrando el origen de datos. De esta forma se liberan las estructuras de datos asociadas a este usuario en el servidor.

Está claro que ahorraríamos más recursos si cerráramos también la conexión de la base de datos. El problema consiste en que la reapertura de la conexión consumiría demasiado tiempo. Los tipos duros pueden programar una especie de caché de conexiones para eliminar la dificultad ... o pueden recurrir a *Microsoft Transaction Server*, que se ocupa precisamente de este tipo de administración.

El método de grabación es también relativamente sencillo:

```

void __fastcall TmodDatos::Save()
{
    Emp->CheckBrowseMode();
    Dept->CheckBrowseMode();
    if (Dept->ChangeCount == 0) return;
    DB->StartTransaction();
    try {
        int ECount;
        OleVariant Delta = Dept->Delta;
        Dept->Reconcile(prDept->ApplyUpdates(Delta, -1, ECount));
        qrEmp->Close();
        if (ECount > 0) Sysutils::Abort();
        DB->Commit();
    }
    catch (Exception&) {
        DB->Rollback();
    }
}

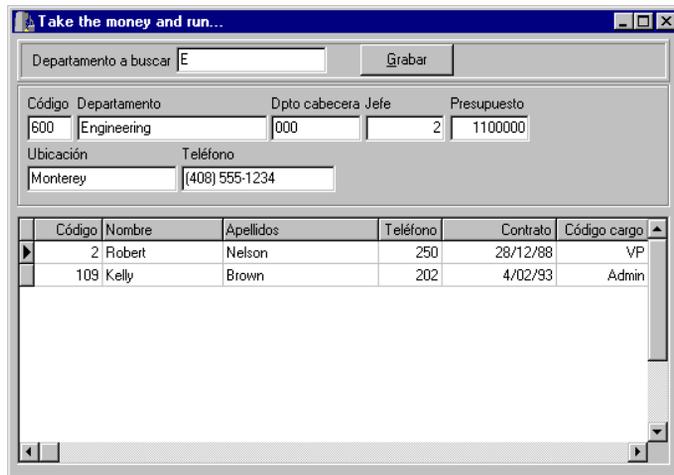
```

```

    throw;
}
}

```

Y ya podemos dedicarnos a traer componentes visuales al formulario principal, como sugiere la siguiente imagen:



Observe, en primer lugar, que no hay una interfaz de navegación en el sentido “tradicional”: primer registro, registro anterior, registro siguiente... Por el contrario, navegamos tecleando prefijos para el nombre del departamento en un cuadro de edición. El evento *OnChange* del editor hace lo siguiente:

```

void __fastcall TwndMain::Edit1Change(TObject *Sender)
{
    Timer1->Enabled = False;
    Timer1->Enabled = True;
}

```

Timer1 es un temporizador con un intervalo de 500 milisegundos, inactivo en tiempo de diseño. Su propósito es realizar la búsqueda medio segundo después de que el usuario haya dejado de teclear:

```

void __fastcall TwndMain::Timer1Timer(TObject *Sender)
{
    modDatos->Load(Edit1->Text + "%");
    Timer1->Enabled = False;
}

```

Y la grabación es trivial, pues se reduce a llamar al método *Save* del módulo de datos. Recuerde solamente que debe manejar el evento *OnReconcileError* del conjunto de datos *Dept*, para informar al usuario de los errores durante la grabación.

Servidores de Internet

PARA SER SINCEROS, TITULAR UN CAPÍTULO tal como lo hemos hecho revela un poco de presuntuosidad. Los temas relacionados con Internet son tantos que pueden redactarse libros completos (los hay). Tenemos el estudio de la infinidad de protocolos existentes, la comunicación mediante *sockets*, el envío y recepción de correo electrónico, el uso de controles ActiveX y las ActiveForms que nos permiten programar clientes de Internet inteligentes...

Sin embargo, este capítulo estará dedicado a un tema muy concreto: la programación de extensiones de servidores HTTP. Se trata de aplicaciones que se ejecutan en el servidor Web al que se conectan los navegadores de Internet, y que permiten el desarrollo de páginas con información dinámica. Tradicionalmente se han utilizado programas desarrollados para la interfaz CGI con este propósito. C++ Builder permite, a partir de la versión 3, desarrollar aplicaciones para la interfaz CGI, así como para las más modernas y eficientes interfaces ISAPI y NSAPI, de Microsoft y Netscape respectivamente. Pero lo más importante de todo es que podemos abstraernos del tipo concreto de interfaz final con la que va a trabajar la extensión. Para ilustrar estas técnicas, desarrollaremos una sencilla aplicación Web de búsqueda de información por palabras claves.

El modelo de interacción en la Web

El protocolo HTTP es un caso particular de arquitectura cliente/servidor. Un cliente, haciendo uso de algún navegador políticamente correcto, pide un “documento” a un servidor situado en determinado punto de la topología de la Internet; por ahora, no nos interesa saber cómo se produce la conexión física entre ambas máquinas. He puesto la palabra “documento” entre comillas porque, como veremos en breve, éste es un concepto con varias interpretaciones posibles.

Una vez que el servidor recibe la petición, envía al cliente el documento solicitado, si es que éste existe. Lo más importante que hay que comprender es lo siguiente: una vez enviado el documento, el servidor se desentiende totalmente del cliente. El servidor nunca sabrá si el cliente estuvo mirando la página uno o quince minutos, o si

cuando terminó siguió o no ese enlace que le habíamos recomendado al final de la página. Esta es la característica más abominable de Internet, inducida por las peculiaridades del hardware y los medios de transporte de la información, pero que complica gravemente la programación de aplicaciones para la red global. El navegante que utiliza Internet principalmente para bajarse imágenes bonitas, excitantes o jocosas, incluye en sus plegarias el que Internet 2 aumente su velocidad de transmisión. Nosotros, los Programadores De Negro, rezamos porque mejore su protocolo infernal.

Para solicitar un “documento” al servidor, el cliente debe indicar qué es lo que desea mediante una cadena conocida como URL: *Uniform Resource Locator*. La primera parte de una URL tiene el propósito de indicar en qué máquina vamos a buscar la página que deseamos, y qué protocolo de comunicación vamos a utilizar:

```
http://www.marteens.com/trucos/truco01.htm
```

El protocolo utilizado será HTTP; podría haber sido, por ejemplo, FILE, para abrir un documento local. La máquina queda identificada por el nombre de dominio:

```
www.marteens.com
```

El resto de la URL anterior describe un directorio dentro de la máquina, y un documento HTML situado dentro de este directorio:

```
trucos/truco01.htm
```

Este directorio es relativo al directorio definido como público por el servidor HTTP instalado en la máquina. En mi portátil, por ejemplo, el directorio público HTTP se llama *webshare*, y ha sido definido por el Personal Web Server de Microsoft, que es el servidor que utilizo para pruebas.

Más adelante veremos que una URL no tiene por qué terminar con el nombre de un fichero estático que contiene texto HTML, sino que puede apuntar a programas y bibliotecas dinámicas, y ahí será donde tendremos la oportunidad de utilizar a nuestro querido C++ Builder.

Aprenda HTML en 14 minutos

No hace falta decirlo por ser de sobra conocido: las páginas Web están escritas en el lenguaje HTML, bastante fácil de aprender sobre todo con una buena referencia al alcance de la mano. HTML sirve para mostrar información, textual y gráfica, con un formato similar al que puede ofrecer un procesador de textos sencillo, pero además permite la navegación de página en página y un modelo simple de interacción con el servidor Web.

Un documento HTML cercano al mínimo sería como el siguiente:

```
<HTML>
<HEAD><TITLE>Mi primer documento</TITLE></HEAD>
<BODY>¡Hola, mundo!</BODY>
</HTML>
```

Este documento mostraría el mensaje *¡Hola, mundo!* en la parte de contenido del navegador, mientras que la barra de título del mismo se cambiaría a *Mi primer documento*. Los códigos de HTML, o *etiquetas*, se encierran entre paréntesis angulares: `<>`. Una gran parte de las etiquetas consiste en pares de apertura y cierre, como `<TITLE>` y `</TITLE>`; la etiqueta de cierre utiliza el mismo nombre que la de apertura, pero precedido por una barra inclinada. Otras etiquetas, como `<HR>`, que dibuja una línea horizontal, realizan su cometido por sí mismas y no vienen por pares.

Existe un amplio repertorio de etiquetas que sirven para dar formato al texto de un documento. Entre éstas tenemos, por ejemplo:

Etiqueta	Significado
<code><H1></code> , <code><H2></code> ...	Encabezamientos
<code><P></code>	Marca de párrafo
<code></code> , <code></code>	Listas ordenadas y sin ordenar
<code></code>	Inclusión de imágenes
<code><PRE></code>	Texto preformateado

Una de las etiquetas más importantes es el *ancla* (*anchor*), que permite navegar a otra página. Por ejemplo:

```
<HTML>
<HEAD><TITLE>Mi primer documento</TITLE></HEAD>
<BODY>
<H1>¡Hola, mundo!</H1>
<HR>
<P>Visite la página de <A HREF=http://www.borland.com>Inprise Corporation</A> para más información sobre C++ Builder.</P>
</BODY>
</HTML>
```

En el ejemplo anterior, la dirección del enlace que se ha suministrado es una URL completa, pero podemos incorporar enlaces locales: dentro del propio servidor donde se encuentra la página actual, e incluso a determinada posición dentro de la propia página. Existen más etiquetas de este tipo, como la etiqueta `<ADDRESS>`, que indica una dirección de correo electrónico. También se puede asociar un enlace a una imagen, colocando una etiqueta `` dentro de un ancla.

El subconjunto de HTML antes descrito ofrece, en cierto sentido, las mismas características del sistema de ayuda de Windows. Es, en definitiva, una herramienta para

definir sistemas de navegación a través de textos (vale, y gráficos también). Pero se necesita algo más si se desea una comunicación mínima desde el cliente hacia el servidor, y ese algo lo proporcionan los *formularios HTML*. Mediante estos formularios, el usuario tiene a su disposición una serie de controles de edición (no muy sofisticados, a decir verdad) que le permiten teclear datos en la pantalla del navegador y remitirlos explícitamente al servidor. Un formulario ocupa una porción de un documento HTML:

```
<HTML>
<HEAD><TITLE>Mi primer documento</TITLE></HEAD>
<BODY>
<H1>¡Hola, mundo!</H1>
<HR>
<FORM METHOD=GET ACTION=http://www.marteens.com/scripts/prg.exe>
<P>Nombre: <INPUT TYPE="TEXT" NAME="Nombre"></P>
<P>Apellidos: <INPUT TYPE="TEXT" NAME="Apellidos"></P>
<INPUT TYPE="SUBMIT" VALUE="Enviar">
</FORM>
</BODY>
</HTML>
```

El aspecto de este documento sobre un navegador es el siguiente:



La etiqueta `<FORM>` marca el inicio del formulario, y define dos parámetros: `METHOD` y `ACTION`. El método casi siempre es `POST` o `GET`, e indica de qué manera se pasan los valores tecleados por el usuario al servidor. Si utilizamos `GET`, los parámetros y sus valores se añaden a la propia URL de destino del formulario; en breve veremos cómo. En cambio, con el método `POST` estos valores se suministran dentro del cuerpo de la petición, como parte del flujo de datos del protocolo `HTTP`.

Extensiones del servidor y páginas dinámicas

La etiqueta `ACTION` indica a qué URL se dirige la petición del cliente. En el ejemplo hemos utilizado esta cadena:

```
http://www.marteens.com/scripts/prg.exe
```

Ya lo habíamos anunciado: una URL no termina necesariamente con el nombre de un fichero HTML. En este caso, termina con el nombre de un programa: *prg.exe*. El propósito de este programa es generar dinámicamente el texto correspondiente a la página HTML que debe enviarse al cliente. Esta aplicación se ejecutará con una serie de parámetros, que se construirán tomando como base los valores suministrados por el cliente al formulario. Por supuesto, el algoritmo de generación de la página tendrá en cuenta estos parámetros.

A este tipo de programas creadores de páginas dinámicas se le conoce como aplicaciones CGI, del inglés *Common Gateway Interface*. Es responsabilidad del servidor HTTP el ejecutar la extensión CGI adecuada a la petición del cliente, y pasarle los parámetros mediante la entrada estándar. El texto generado por el programa se envía de vuelta al servidor HTTP a través de la salida estándar de la aplicación, un mecanismo muy a lo UNIX.

Como puede imaginar, todo este proceso de llamar al ejecutable, configurar el flujo de entrada y recibir el flujo de salida es un proceso costoso para el servidor HTTP. Además, si varios usuarios solicitan simultáneamente los servicios de una aplicación CGI, esta será cargada en memoria tantas veces como sea necesario, con el deducible desperdicio de recursos del ordenador. Los dos principales fabricantes de servidores HTTP en el mundo de Windows, Microsoft y Netscape, idearon mecanismos equivalentes a CGI, pero que utilizan DLLs en vez de ficheros ejecutables. La interfaz de Microsoft se conoce como ISAPI, mientras que la de Netscape es la NSAPI, y éstas son algunas de sus ventajas:

- El proceso de carga es más rápido, porque el servidor puede mantener cierto número de DLLs en memoria un tiempo determinado, de modo que una segunda petición de la misma extensión no necesite volver a cargarla.
- La comunicación es más rápida, porque el servidor ejecuta una función de la DLL, pasando los datos y recibiendo la página mediante parámetros en memoria.
- La ejecución concurrente de la misma extensión debido a peticiones simultáneas de cliente es menos costosa en recursos, porque el código está cargado una sola vez.

¿Desventajas? Hay una importante, sobre todo para los malos programadores (es decir, que no nos afecta):

- La extensión pasa a formar parte del espacio de procesos del servidor. Si se cuelga, el servidor (o uno de sus hilos) también se cuelga.

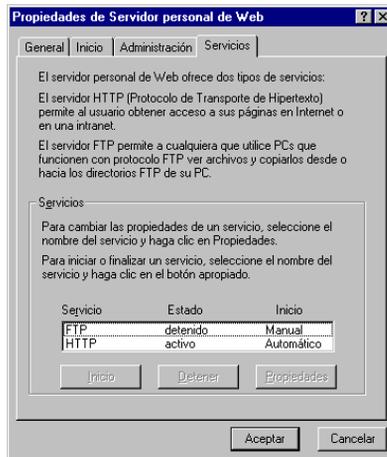
Nos queda pendiente explicar cómo se pasan los parámetros a la extensión del servidor. Supongamos que el usuario teclea, en el formulario que hemos utilizado como ejemplo, su nombre y apellidos, y pulsa el botón para enviar la solicitud. Como el método asociado al formulario es GET, la URL que se buscará tendrá la siguiente forma completa:

```
http://www.marteens.com/scripts/  
prg.exe?Nombre=Ian&Apellidos=Marteens
```

Si el método fuera POST, los datos que aparecen después del signo de interrogación se codificarían dentro del propio protocolo HTTP. Pero en el fondo se estaría pasando la misma información que en el otro caso.

¿Qué necesito para este seguir los ejemplos?

Para desarrollar extensiones de Web con C++ Builder va a necesitar una copia de C++ Builder ... pero creo que esto ya se lo imaginaba. Además, hace falta un explorador de Internet, de la marca que sea y un servidor HTTP. Aquí es donde pueden complicarse un poco las cosas. Veamos, ¿tiene usted un Windows NT? Más bien, ¿tiene un BackOffice, de Microsoft? Entonces tendrá un Internet Information Server, o podrá instalarlo si todavía no lo ha hecho. ¿Y qué pasa si el lector pertenece a la secta que clava alfileres en el mapa en el sitio donde dice Redmond? Siempre puede utilizar el servidor de la competencia, el Netscape FastTrack.



Otra solución es utilizar Personal Web Server para Windows 95, un producto que puede conseguirse gratuitamente en la página Web de Microsoft. O, si tiene Windows 98, aprovechar que esta versión ya lo trae incorporado. La imagen anterior muestra una de las páginas del diálogo de propiedades del servidor, que puede ejecutarse

desde el Panel de Control, o desde el icono en la Bandeja de Iconos, si se ha cargado este programa.

Si va a desarrollar extensiones ISAPI/NSAPI en vez de CGI, se encontrará con que necesitará descargar la DLL programada cada vez que quiera volver a compilar el proyecto, si es que la ha probado con un navegador. Lo que sucede es que el servidor de Internet deja a la DLL cargada en una caché, de forma tal que la próxima petición a esa URL pueda tener una respuesta más rápida. Mientras la DLL esté cargada, el fichero en disco estará bloqueado, y no podrá ser sustituido. Este comportamiento es positivo una vez que nuestra aplicación esté en explotación, pero es un engorro mientras la desarrollamos. Una solución es detener cada vez el servidor después de cada cambio, lo cual es factible solamente cuando lo estamos probando en modo local. La mejor idea es desactivar temporalmente la caché de extensiones en el servidor Web, modificando la siguiente clave del registro de Windows:

```
[HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\W3Svc\
Parameters]
CacheExtensions="00"
```

El ejemplo mostrado en este capítulo ha sido desarrollado y depurado utilizando Personal Web Server de Microsoft para Windows 95.

Módulos Web

Podemos crear aplicaciones CGI fácilmente con C++ Builder: basta con desarrollar una aplicación de tipo *consola* y manejar directamente la entrada estándar y la salida estándar. Recuerde que las aplicaciones de tipo consola se crean en el diálogo de opciones del proyecto, y que son aproximadamente equivalentes a las aplicaciones “tradicionales” de MS-DOS. Desde siempre hemos podido generar este estilo de aplicaciones. Pero seríamos responsables de todo el proceso de análisis de la URL, de los parámetros recibidos y de la generación del código HTML con todos sus detalles. Si nuestra extensión Web es sencilla, como puede ser implementar un contador de accesos, no sería demasiado asumir todas estas tareas, pero las aplicaciones más complejas pueden irsenos de las manos. Lo mismo es aplicable a la posibilidad de programar directamente DLLs para las interfaces ISAPI y NSAPI: tarea reservada para tipos duros de matar.

Para crear una aplicación Web, debemos invocar al Depósito de Objetos, comando *File|New*, y elegir el icono *Web Server Application*. El diálogo que aparece a continuación nos permite especificar qué modelo de extensión Web deseamos:



En cualquiera de los casos, aparecerá en pantalla un módulo de datos, de nombre *WebModule1* y perteneciente a una clase derivada de *TWebModule*. Lo que varía es el fichero de proyecto asociado. Si elegimos crear una aplicación CGI, C++ Builder generará el siguiente fichero *dpr*:

```
//-----
#include <condefs.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <SysUtils.hpp>
#include <httpapp.hpp>
#include <CGIApp.hpp>
#pragma hdrstop
USEFORM("Unit1.cpp", WebModule1); /* TWebModule: DesignClass */
//-----
#define Application Httpapp::Application
#pragma link "cgiapp.obj"
//-----
int main(int argc, char* argv[])
{
    try
    {
        Application->Initialize();
        Application->CreateForm(__classid(TWebModule1), &WebModule1);
        Application->Run();
    }
    catch (Exception &exception)
    {
        Sysutils::ShowException(&exception, Sysutils::ExceptAddr());
    }
    return 0;
}
```

Como se puede ver, el proyecto será compilado como un fichero ejecutable. En cambio, si elegimos ISAPI/NSAPI, C++ Builder crea código para una DLL:

```
//-----
#include <HTTPApp.hpp>
#include <ISAPIApp.hpp>
#include <Isapi2.hpp>
#pragma hdrstop
USEFORM("Unit1.cpp", WebModule1); /* TWebModule: DesignClass */
```

```

//-----
#define Application Httpapp::Application
#pragma link "isapiapp.obj"
//-----
int WINAPI DllEntryPoint(HINSTANCE hinst, unsigned long reason,
void*)
{
    try
    {
        Application->Initialize();
        Application->CreateForm(__classid(TWebModule1), &WebModule1);
        Application->Run();
    }
    catch (Exception &exception)
    {}
    return 1;
}
//-----
extern "C"
{
    BOOL __export WINAPI GetExtensionVersion(
        Isapi2::THSE_VERSION_INFO &Ver)
    {
        return Isapiapp::GetExtensionVersion(Ver);
    }
    //-----
    int __export WINAPI HttpExtensionProc(
        Isapi2::TEXTENSION_CONTROL_BLOCK &ECB)
    {
        return Isapiapp::HttpExtensionProc(ECB);
    }
    //-----
    BOOL __export WINAPI TerminateExtension(int dwFlags)
    {
        return Isapiapp::TerminateExtension(dwFlags);
    }
}

```

¿Tres funciones para exportar? Sí, se trata de las funciones que el servidor Web espera que tenga nuestra extensión, y las llamará para la carga de la DLL, la generación de documentos dinámicos y para descargar finalmente la DLL cuando haga falta. C++ Builder implementa estas tres funciones por nosotros.

Ahora seguramente el lector saltará de alegría: le voy a contar cómo corregir un *bug* del asistente de generación de aplicaciones ISAPI/NSAPI de Borland. Resulta que existe una variable global en la unidad *System* que es utilizada por el administrador de memoria dinámica de C++ Builder para saber si puede utilizar o no el algoritmo más rápido, pero no reentrante, que asume la existencia de un solo subproceso dentro de la aplicación. Esta variable se llama *IsMultiThread*, y tenemos que añadir la siguiente inicialización en el fichero de proyecto:

```
IsMultiThread = True;
```

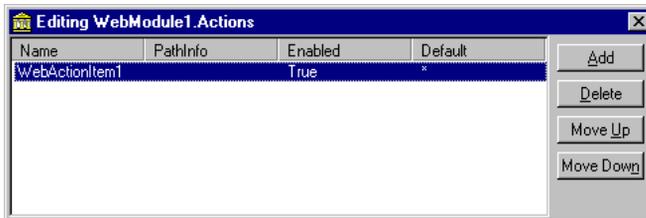
C++ Builder 4 ha corregido el *bug*, inicializando *IsMultiThread* en las unidades de soporte de las extensiones ISAPI/NSAPI.

Existe una alternativa a la creación de nuevos módulos Web mediante el Asistente de C++ Builder. Si ya tenemos un módulo de datos, desarrollado en otra aplicación, podemos crear el esqueleto inicial tal como hemos explicado, eliminar del proyecto el módulo Web y añadir entonces el módulo existente. ¡Pero el módulo eliminado descendiende de *TWebModule*, mientras que el módulo añadido es un vulgar descendiente de *TDataModule*! No importa, porque a continuación añadiremos al módulo un componente *TWebDispatcher*, de la página *Internet*. Este componente contiene una propiedad llamada *Actions*, que es precisamente la que marca la diferencia entre los módulos de datos “normales” y los módulos Web.

Acciones

Cada vez que un navegante solicita un documento dinámico a una extensión de servidor, se ejecuta la correspondiente aplicación CGI, o la función diseñada para este propósito de la DLL ISAPI ó NSAPI. Sin embargo, en cada caso los parámetros de la petición contenidos en la URL pueden ser distintos. El primer parámetro verdadero contenido en la URL va a continuación del nombre del módulo, separado por una barra inclinada, y se conoce como *información de camino*, o *path info*. Y para discriminar fácilmente entre los valores de este parámetro especial los módulos Web de C++ Builder ofrecen las *acciones*.

Tanto *TWebModule* como *TWebDispatcher* contienen una propiedad *Actions*, que es una colección de elementos de tipo *TWeb.ActionItem*. La siguiente imagen muestra el editor de esta propiedad:



Cada objeto *Web.ActionItem* que se añade a esta colección posee las siguientes propiedades importantes:

Propiedad	Significado
<i>PathInfo</i>	El nombre del camino especificado en la URL, después del nombre de la aplicación

Propiedad	Significado
<i>MethodType</i>	El tipo de petición ante la cual reacciona el elemento
<i>Default</i>	Si es <i>True</i> , se dispara cuando no se encuentre una acción más apropiada
<i>Enabled</i>	Si está activo o no

Estos objetos tienen un único evento, *OnAction*, que es el que se dispara cuando el módulo Web determina que la acción es aplicable, y su tipo es éste:

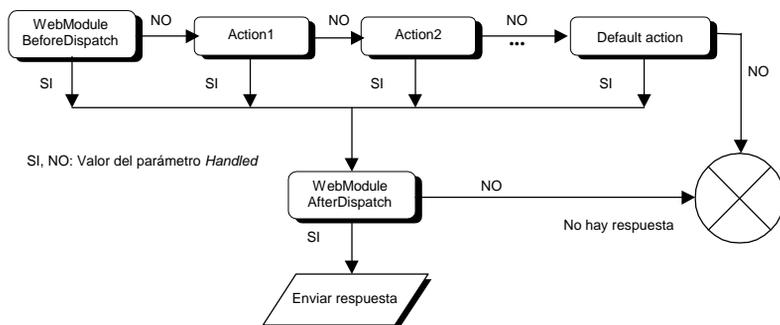
```
typedef void __fastcall (__closure *THTTPMethodEvent)
(Object* Sender, TWebRequest *Request, TWebResponse *Response,
bool &Handled);
```

Toda la información acerca de la petición realizada por el cliente viene en el parámetro *Request* del evento, que analizaremos en la próxima sección. El propósito principal de los manejadores de este evento es asignar total o parcialmente el texto HTML que debe enviarse al cliente como respuesta, en el parámetro *Response*. Digo parcialmente porque para una misma petición pueden dispararse varias acciones en cascada, como veremos enseguida; cada acción puede construir una sección de la página HTML final. Pero, además, en la respuesta al evento asociado a la acción se pueden provocar efectos secundarios, como grabar o modificar un registro en una base de datos, alterar variables del módulo Web, y cosas parecidas.

Supongamos que el cliente pide la siguiente URL:

```
http://www.wet_wild_woods.com/scripts/buscar.dll/clientes
```

La información de ruta viene a continuación del nombre de la DLL: *clientes*. Cuando nuestra aplicación recibe la petición, busca todas las acciones que tienen asignada esta cadena en su propiedad *PathInfo*. El siguiente diagrama muestra la secuencia de disparo de las distintas acciones dentro del módulo:



Antes de comenzar a recorrer las posibles acciones, el módulo Web dispara su propio evento *BeforeDispatch*, cuyo tipo es idéntico al de *OnAction*. De este modo, el módulo

tiene la oportunidad de preparar las condiciones para la cadena de acciones que puede dispararse a continuación. El evento utiliza un parámetro *Handled*, de tipo lógico. Si se le asigna *True* a este parámetro y se asigna una respuesta a *Response* (paciencia, ya contaré cómo), no llegarán a dispararse el resto de las acciones del módulo, y se devolverá directamente la respuesta asignada.

En caso contrario, el módulo explora secuencialmente todas las acciones cuya propiedad *PathInfo* sea igual a *clientes*; pueden existir varias. Para que la cadena de disparos no se interrumpa, cada acción debe dejar el valor *False* en el parámetro *Handled* del evento. Si después de probar todas las acciones que corresponden por el valor almacenado en su *PathInfo*, ninguna ha marcado *Handled* como verdadera (o no se ha encontrado ninguna), se trata de ejecutar aquella acción que tenga su propiedad *Default* igual a *True*, si es que hay alguna.

Al finalizar todo este proceso, y si alguna acción ha marcado como manejada la petición, se ejecuta el evento *AfterDispatch* del módulo Web.

Recuperación de parámetros

¿Dónde vienen los parámetros de la petición? Evidentemente, en las propiedades del parámetro *Request* del evento asociado a la acción. La propiedad concreta en la que tenemos que buscarlos depende del tipo de acción que recibimos. Claro está, si no conocemos qué tipo de método corresponde a la acción, tenemos que verificar la propiedad *MethodType* de la petición, que pertenece al siguiente tipo enumerativo:

```
enum TMethodType { mtAny, mtGet, mtPut, mtPost, mtHead };
```

Estamos interesados principalmente en los tipos *mtGet* y *mtPost*. El tipo *mtAny* representa un comodín, incluso para otros tipos de métodos menos comunes no incluidos en *TMethodType*, como *OPTIONS*, *DELETE* y *TRACE*; en estos casos, hay que mirar también la propiedad *Method*, que contiene la descripción literal del método empleado.

Supongamos ahora que el método sea *GET*. La cadena con todos los parámetros viene en la propiedad *Query* de *Request*. Por ejemplo:

```
URL: http://www.WetWildWoods.com/find.dll/animal?kind=cat&walk=alone
Query: kind=cat&walk=alone
```

Pero más fácil es examinar los parámetros individualmente, mediante la propiedad *QueryFields*, de tipo *TStrings*. En el siguiente ejemplo se aprovecha la propiedad vectorial *Values* de esta clase para aislar el nombre del parámetro del valor:

```

void __fastcall TWebModule1::WebModule1WebActionItem1Action(
    TObject *Sender, TWebRequest *Request, TWebResponse *Response,
    bool &Handled)
{
    if (Request->QueryFields->Values["kind"] == "cat")
        // ... están preguntando por un gato ...
}

```

Por el contrario, si el método es POST los parámetros vienen en las propiedades *Content*, que contiene todos los parámetros sin descomponer, y en *ContentFields*, que es análoga a *QueryFields*. Si el programador no quiere depender del tipo particular de método de la acción para el análisis de parámetros, puede utilizar uno de los procedimientos auxiliares *ExtractContentFields* ó *ExtractQueryFields*:

```

void __fastcall TWebModule1::WebModule1WebActionItem1Action(
    TObject *Sender, TWebRequest *Request, TWebResponse *Response,
    bool &Handled)
{
    if (Request->MethodType == mtPost)
        Request->ExtractContentFields(Request->QueryFields);
    if (Request->QueryFields->Values["kind"] == "cat")
        // ... están preguntando por un gato ...
}

```

Generadores de contenido

El propósito de los manejadores de eventos de acciones es, principalmente, generar el contenido, o parte del contenido de una página HTML. Esto se realiza asignando el texto HTML a la propiedad *Content* del parámetro *Response* del evento en cuestión. Por ejemplo:

```

void __fastcall TWebModule1::WebModule1WebActionItem1Action(
    TObject *Sender, TWebRequest *Request, TWebResponse *Response,
    bool &Handled)
{
    Response->Content = "<HTML><BODY>;Hola, colega!</BODY></HTML>";
}

```

Por supuesto, siempre podríamos asignar una larga cadena generada por código a esta propiedad, siempre que contenga una página HTML válida. Pero vemos que, incluso en el sencillo ejemplo anterior, ésta es una tarea pesada y es fácil cometer errores de sintaxis. Por lo tanto, C++ Builder nos ofrece varios componentes en la página *Internet* para facilitar la generación de texto HTML. Para empezar, veamos el componente *TPageProducer*, cuyas propiedades relevantes son las siguientes:

Propiedad	Significado
<i>Dispatcher</i>	El módulo en que se encuentra, o el componente <i>TWebDispatcher</i> al que se asocia

Propiedad	Significado
<i>HTMLDoc</i>	Lista de cadenas que contiene texto HTML
<i>HTMLFile</i>	Alternativamente, un fichero con texto HTML

La idea es que *HTMLDoc* contenga el texto a generar por el componente, de forma tal que este texto se especifica en tiempo de diseño y se guarda en el fichero *dfm* del módulo, para que no se mezcle con el código. Pero también puede especificarse un fichero externo en *HTMLFile*, para que pueda modificarse el texto generado sin necesidad de tocar el ejecutable de la aplicación. En cualquiera de estos dos casos, el ejemplo de respuesta al evento anterior se escribiría de este otro modo:

```
void __fastcall TWebModule1::WebModule1WebActionItem1Action(
    TObject *Sender, TWebRequest *Request, TWebResponse *Response,
    bool &Handled)
{
    Response->Content = PageProducer1->Content();
}
```

No obstante, si nos limitamos a lo que hemos descrito, las páginas producidas por nuestra extensión de servidor siempre serán páginas estáticas. La principal ventaja del uso de *TPageProducer* es que podemos realizar la sustitución dinámica de *etiquetas transparentes* por texto. Una etiqueta transparente es una etiqueta HTML cuyo primer carácter es la almohadilla: #. Estas etiquetas no pertenecen en realidad al lenguaje, y son ignoradas por los navegadores. En el siguiente ejemplo, el propósito de la etiqueta `<#HORA>` es el de servir como comodín para ser sustituido por la hora actual:

```
<HTML><BODY>
¡Hola, colega! ¿Qué haces por aquí a las <#HORA>?
</BODY></HTML>
```

¿Por qué se sustituye la etiqueta transparente anterior por la hora? ¿Acaso hay algún mecanismo automático que detecte el nombre de la etiqueta y ...? No, por supuesto. Cuando utilizamos la propiedad *Content* del productor de páginas, estamos iniciando en realidad un algoritmo en el que nuestro componente va examinando el texto HTML que le hemos suministrado, va detectando las etiquetas transparentes y, para cada una de ellas, dispara un evento durante el cual tenemos la posibilidad de indicar la cadena que la sustituirá. Este es el evento *OnHTMLTag*, y el siguiente ejemplo muestra sus parámetros:

```
void __fastcall TmodWebData::PageProducer1HTMLTag(TObject *Sender,
    TTag Tag, const AnsiString TagString, TString *TagParams,
    AnsiString &ReplaceText)
{
    if (TagString == "HORA")
        ReplaceText = TimeToStr(Now());
}
```

Generadores de tablas

Un caso particular de generadores de contenido son los generadores de tablas que incorpora C++ Builder: *TDataSetTableProducer*, y *TQueryTableProducer*. Ambos son muy parecidos, pues descienden de la clase abstracta *TDSTableProducer*, y generan una tabla HTML a partir de los datos contenidos en un conjunto de datos de la VCL. *TQueryTableProducer* se diferencia en que el conjunto de datos debe ser obligatoriamente una consulta, y en que esta consulta puede extraer sus parámetros de los parámetros de la petición HTTP en curso, ya sea a partir de *QueryFields*, en el caso de acciones GET, o de *ContentFields*, en el caso de POST.

Las principales propiedades comunes a estos componentes son:

Propiedad	Significado
<i>DataSet</i>	El conjunto de datos asociado
<i>MaxRows</i>	Número máximo de filas de datos generadas
<i>Caption, CaptionAlignment</i>	Permite añadir un título a la tabla
<i>Header, Footer</i>	Para colocar texto antes y después de la tabla
<i>Columns, RowAttributes, ColumnAttributes</i>	Formato de la tabla

Para dar formato a la tabla, es conveniente utilizar el editor asociado a la propiedad *Columns* de estos componentes. La siguiente imagen muestra el aspecto del editor de columnas del productor de tablas, que permite personalizar columna por columna el aspecto del código generado:



Para ambos componentes, la función *Content* genera el texto HTML correspondiente. En el caso de *TQueryTableProducer*, la propia función se encarga de extraer los parámetros de la solicitud activa, y de asignarlos al objeto *TQuery* asociado.

El evento *OnFormatCell*, común a ambos componentes, es muy útil, porque permite retocar el contenido de cada celda de la tabla generada. Podemos cambiar colores,

alineación, tipos de letras, e incluso sustituir completamente el contenido. Por ejemplo, el siguiente ejemplo muestra cómo se pueden generar cuadros de edición en la primera columna de una tabla:

```
void __fastcall TForm1::DataSetTableProducer1FormatCell(
    TObject *Sender, int CellRow, int CellColumn,
    THTMLBgColor &BgColor, THTMLAlign &Align, THTMLVAlign &VAlign,
    AnsiString &CustomAttrs, AnsiString &CellData)
{
    if (CellRow > 0 && CellColumn == 0)
        CellData = "<INPUT TYPE=\"TEXT\" VALUE=" +
            AnsiQuotedStr(CellData, '\\\"') + ">";
}
```

Mantenimiento de la información de estado

El problema principal de los servidores Web es su corta memoria. Usted le pide a uno de ellos: dame, por favor, la lista de los diez discos más vendidos, y el servidor le responderá con mil amores. Pero si se le ocurre preguntar por los diez que siguen, el servidor fruncirá una ceja: ¿y quién eres tú?

Evidentemente, no tiene sentido que el servidor recuerde la conversación que ha mantenido con nosotros. Después que leamos la página que nos ha enviado la primera vez como respuesta, es muy probable que apaguemos el navegador, o nos vayamos a navegar a otra parte. No merece la pena guardar memoria de los potenciales cientos o miles de usuarios que pueden conectarse diariamente a una página muy transitada.

Pero no hay problemas insolubles, sino preguntas mal planteadas. En nuestro ejemplo anterior, acerca de la lista de éxitos, podemos formular la petición de este otro modo: ¿cuáles son los discos de la lista que van desde el 11 al 20? O en esta otra forma: hola, soy Ian Marteens (conexión número 12345678), ¿cuáles son los próximos diez discos? En este último caso, por ejemplo, necesitamos que se cumplan estas dos condiciones:

- El servidor debe asignar a cada conexión una identificación, del tipo que sea. Además, debe llevar en una base de datos un registro de las acciones realizadas por la “conexión”.
- El usuario debe disponer a su vez de este identificador, lo que implica que el servidor debe pensar en algún método para comunicar este número al cliente.

Sigamos aclarando el asunto: observe que yo, Ian Marteens, puedo ser ahora la conexión 3448 para cierto servidor, pero al apagar el ordenador y volver a conectarme al día siguiente, recibiré el número 5237. Por supuesto, podemos idear algún meca-

nismo de identificación que nos permita recuperar nuestra última identificación, pero esta es una variación sencilla del mecanismo básico que estamos explicando.

¿Cómo puede comunicar el servidor al cliente su número de identificación? Planteémoslo de otra manera: ¿qué es lo que un servidor de Internet puede suministrar a un cliente? ¡Documentos HTML, qué diablos! Pues bien, introduzca traicioneramente el identificador de la conexión dentro del documento HTML que se envía como respuesta. Se supone que se recibe un número de conexión porque queremos seguir preguntando tonterías al servidor (bueno, ¿y qué?). Entonces es muy probable que el documento contenga un formulario, y que podamos utilizar un tipo especial de campo conocido como *campos ocultos* (*hidden fields*):

```
<HTML>
<HEAD><TITLE>Canciones más solicitadas</TITLE></HEAD>
<BODY>
<H1>Lista de éxitos</H1>
<H2>(del 1 al 5)</H2>
<HR><OL START=1><LI>Dust in the wind</LI>
<LI>Stairway to heaven</LI>
<LI>More than a feeling</LI>
<LI>Wish you were here</LI>
<LI>Macarena (uh-oh)</LI></OL><HR>
<FORM METHOD=GET
  ACTION=http://www.marteens.com/scripts/prg.exe/hits>
<INPUT TYPE="HIDDEN" NAME="USER" VALUE="1234">
<INPUT TYPE="SUBMIT" VALUE="Del 6 al 10">
</FORM>
</BODY>
</HTML>
```

De todo el documento anterior, la cláusula que nos interesa es la siguiente:

```
<INPUT TYPE="HIDDEN" NAME="USER" VALUE="1234">
```

Esta cláusula no genera ningún efecto visual, pero como forma parte del cuerpo del formulario, contribuye a la generación de parámetros cuando se pulsa el botón de envío. El atributo `NAME` vale `USER` en este ejemplo, pero podemos utilizar un nombre arbitrario para el parámetro. Como el método del formulario es `GET`, la pulsación del botón solicita la siguiente URL:

```
http://www.marteens.com/scripts/prg.exe/hits?USER=1234
```

Cuando el servidor Web recibe esta petición puede buscar en una base de datos cuál ha sido el último rango de valores enviado al usuario 1234, generar la página con los próximos valores, actualizar la base de datos y enviar el resultado. Otra forma de abordar el asunto sería incluir el próximo rango de valores dentro de la página:

```
<INPUT TYPE="HIDDEN" NAME="STARTFROM" VALUE="6">
```

De esta forma, no es necesario que el servidor tenga que almacenar en una base de datos toda la actividad generada por una conexión. Este enfoque, no obstante, es mejor utilizarlo en casos sencillos como el que estamos exponiendo.

¿Le apetece una galleta?

Existen muchas otras formas de mantener la información de estado relacionada con un cliente. Una de estas técnicas son las “famosas” *cookies*, que son plenamente soportadas por C++ Builder mediante propiedades y métodos de las clases *TWebResponse* y *TWebRequest*.

Todo comienza cuando una extensión HTTP envía junto a una página HTML una de estas *cookies*. El método de envío pertenece a la clase *TWebResponse*, y tiene el siguiente aspecto:

```
void __fastcall TWebResponse::SetCookieField(TStrings *Values,
      const AnsiString ADomain, const AnsiString APath,
      TDateTime AExpires, bool ASecure);
```

La información básica de este método se pasa en el parámetro *Values*, como una lista de cadenas con el formato *Parámetro=Valor*. Por ejemplo:

```
void __fastcall EnviarIdentificadorUsuario(
      TWebResponse *Response, int UserID)
{
    TStrings *valores = new TStringList;
    valores->Add("UserID=" + IntToStr(UserID));
    Response->SetCookieField(valores, "marteens.com", "",
        Now() + 30, False);
    delete valores;
}
```

El objetivo de una *cookie* es que sea devuelta en algún momento al servidor. Los parámetros *ADomain* y *APath* indican a qué dominio y ruta se debe enviar de vuelta esta información. *AExpires* establece una fecha de caducidad para la *cookie*; en el ejemplo anterior, a partir de treinta días desde la fecha actual. Por último, el parámetro *ASecure*, cuando vale *True*, especifica que solamente puede devolverse la información de la *cookie* a través de una conexión segura.

La otra cara de la moneda: cómo recibir esta información de vuelta. Para eso, la clase *TWebRequest* tiene las propiedades *Cookie*, de tipo *AnsiString*, y *CookieFields*, que apunta a una lista de cadenas. El trabajo con estas propiedades es similar al de las propiedades *Query/QueryFields* y *Content/ContentFields*.

La diferencia principal entre las *cookies* y el uso de parámetros en la consulta o en el contenido consiste en que la información almacenada por el primer mecanismo

puede recuperarse de una sesión a otra, mientras que los restantes sistemas de mantenimiento de información solamente sirven para encadenar secuencias de página durante una misma sesión.

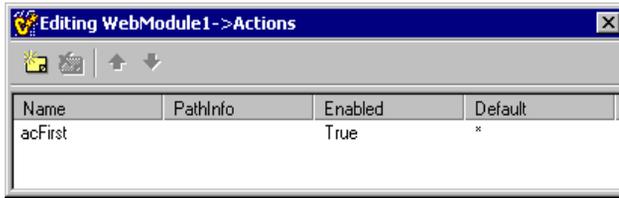
Sin embargo, a pesar de que esta técnica es inofensiva y relativamente poco intrusiva, goza de muy poca popularidad entre los usuarios de la Internet. Cuando una *cookie* es recibida por un navegador Web, se le pregunta al usuario si desea o no aceptarla, y es muy probable que la rechace. También hay que contar con la fecha de caducidad, y con la posibilidad de que el propietario del ordenador se harte de tener la mitad de su disco duro ocupada por porquerías bajadas desde la Internet, y borre todo el directorio de archivos temporales, *cookies* incluidas. Por lo tanto, utilice este recurso para guardar información opcional, de la que pueda prescindir sin mayor problema.

Un simple navegador

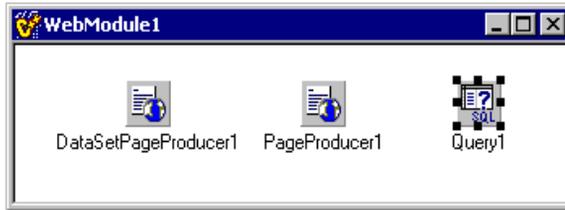
Es conveniente organizar todo el contenido que hemos expuesto mediante un ejemplo sencillo. ¿Qué tal si le muestro cómo navegar registro a registro sobre una tabla alcanzable desde el servidor HTTP? Mostrar un registro de una tabla en una página HTML será muy sencillo, gracias al componente *TDataSetPageProducer*. Lo más complicado será coordinar al cliente y al servidor, de modo que cuando el cliente pida el “próximo” registro, el servidor sepa de qué registro está hablando. La técnica que voy a emplear está inspirada en la forma en el que el BDE implementa la navegación sobre tablas cliente/servidor.

Iniciamos entonces una nueva aplicación Web, mediante el icono *Web Server Application* del Depósito de Objetos; da lo mismo que sea una DLL o un ejecutable, pero le recomiendo que utilice una DLL para que no tenga problemas al seguir el ejemplo. Guarde entonces el proyecto como *WebBrowse*, y la unidad del módulo de datos como *Datos*.

Nuestra aplicación solamente implementará una acción, a la cual no asociaremos nombre alguno. Para controlar el código HTML que genera el servidor utilizaremos un par de parámetros: *Dirección* y *Código*. En el primero indicaremos en qué dirección desea navegar el usuario: *First*, *Prior*, *Next* o *Last*. Cuando se trata de *Prior* o *Next* hay que indicarle al servidor en qué registro se encontraba el usuario, para lo cual utilizaremos el parámetro *Código*. Por lo tanto, pulse sobre la propiedad *Actions* del módulo Web, y añada una nueva acción, de nombre *acFirst*, marcada como *Default*, y deje vacía su propiedad *PathInfo*:



Sobre el módulo Web generado añadiremos los siguientes componentes:



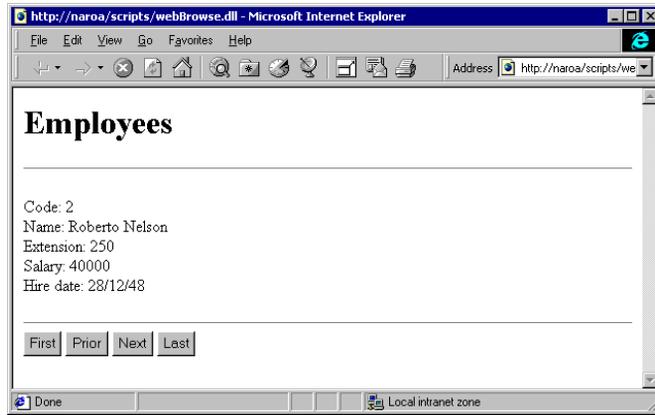
Más sencillo imposible: dos productores de datos HTML (*DataSetPageProducer1* y *PageProducer1*), más una consulta. El componente *DataSetPageProducer1* tiene una propiedad llamada *DataSet*, que debemos hacer que apunte a *Query1*. Por su parte, solamente tenemos que asignar *DatabaseName* a la consulta. He utilizado el alias *bdemos*, pero usted puede utilizar la base de datos que se le antoje. La instrucción SQL de la consulta se generará dinámicamente, cada vez que se realice una petición de página al servidor.

El contenido de *DataSetPageProducer1* se asigna en la propiedad *HTMLDoc*, y será el siguiente:

```
<HTML><BODY>
<H1>Employees</H1><HR>
<FORM METHOD=GET ACTION="WebBrowse.dll">
<P>
Code: <#EmpNo><BR>
Name: <#FirstName> <#LastName><BR>
Extension: <#PhoneExt><BR>
Salary: <#Salary><BR>
Hire date: <#HireDate>
</P><HR>
<#CODE>
<INPUT TYPE="SUBMIT" NAME="Direction" VALUE="First">
<INPUT TYPE="SUBMIT" NAME="Direction" VALUE="Prior">
<INPUT TYPE="SUBMIT" NAME="Direction" VALUE="Next">
<INPUT TYPE="SUBMIT" NAME="Direction" VALUE="Last">
</FORM></BODY></HTML>
```

Observe la presencia de etiquetas con los nombres de campos de la tabla de empleados. Cuando le pidamos a este generador HTML su contenido, automáticamente se sustituirán dichas etiquetas por el valor de los campos del conjunto de datos asociado. Además de las etiquetas que corresponden directamente a campos, he incluido

una etiqueta especial, `<#CODE>`, para que la página “recuerde” cuál es el código del registro visualizado. Este `<#CODE>` será expandido como un campo de formulario de tipo `HIDDEN`. El aspecto de la página generada será el siguiente:



Por su parte, el contenido de *PageProducer1* se mostrará únicamente en una situación muy especial, como veremos más adelante:

```
<HTML><BODY>
<H1>Employees</H1><HR>
<FORM METHOD=GET ACTION="WebBrowse.dll">
<P>Record/key deleted.</P><HR>
<INPUT TYPE="SUBMIT" NAME="Direction" VALUE="First">
<INPUT TYPE="SUBMIT" NAME="Direction" VALUE="Last">
</FORM></BODY></HTML>
```

Después hay que interceptar el evento *OnAction* de la única acción del módulo, para que se ejecute durante el mismo el algoritmo de generación y apertura de la consulta:

```
void __fastcall TWebModule1::WebModule1acFirstAction(
    TObject *Sender, TWebRequest *Request, TWebResponse *Response,
    bool &Handled)
{
    AnsiString Direction;
    int CurrRecord = 0;
    // Extraer parámetros
    if (Request->QueryFields->IndexOfName("CODE") != -1)
    {
        CurrRecord = StrToInt(Request->QueryFields->Values["CODE"]);
        Direction =
            Request->QueryFields->Values["Direction"].UpperCase();
    }
    // Generar contenido
    if (Direction == "FIRST" || Direction == "")
        GenerateQuery("order by EmpNo asc", CurrRecord);
    else if (Direction == "PRIOR")
        GenerateQuery("where EmpNo < %d order by EmpNo desc",
            CurrRecord);
}
```

```

else if (Direction == "NEXT")
    GenerateQuery("where EmpNo > %d order by EmpNo asc",
        CurrRecord);
else
    GenerateQuery("order by EmpNo desc", CurrRecord);
if (Query1->Eof)
    GenerateQuery("where EmpNo = %d", CurrRecord);
if (Query1->Eof)
    Response->Content = PageProducer1->Content();
else
    Response->Content = DataSetPageProducer1->Content();
Query1->Close();
}

```

Quiero que preste atención a esta parte del método:

```

if (Query1->Eof)
    GenerateQuery("where EmpNo = %d", CurrRecord);
if (Query1->Eof)
    Response->Content = PageProducer1->Content();
else
    Response->Content = DataSetPageProducer1->Content();

```

Supongamos que estamos ya en el último registro de la tabla, y que el usuario pide el siguiente. Es evidente que *Query1* estará vacía, porque estaremos solicitando el registro cuyo código es mayor que el mayor de los códigos. En tal caso, generamos una nueva consulta que pida el mismo registro que tenía el usuario antes. Normalmente, esta petición no falla, pero puede darse el caso en que algún proceso concurrente elimine al empleado deseado. Para cubrirnos las espaldas, mostramos entonces la página contenida en *PageProducer1*, que contiene el mensaje de error *Record or key deleted*, y dejamos al usuario la libertad de dirigirse al primer o al último registro de la tabla.

Todo el algoritmo anterior está basado en el uso de un método auxiliar, *GenerateQuery*, que se implementa del siguiente modo:

```

void __fastcall TWebModule1::GenerateQuery(const AnsiString tail,
    int CurrRecord)
{
    Query1->Close();
    // Generar contenido
    Query1->SQL->Clear();
    Query1->SQL->Add("select * from Employee");
    Query1->SQL->Add(Format(tail, ARRAYOFCONST((CurrRecord))));
    Query1->Open();
}

```

Lo único que nos queda es la sustitución de etiquetas durante la generación del contenido de *DataSetPageProducer1*:

```

void __fastcall TWebModule1::DataSetPageProducer1HTMLTag(
    TObject *Sender, TTag Tag, const AnsiString TagString,
    TString *TagParams, AnsiString &ReplaceText)

```

```

{
    if (TagString == "CODE")
        ReplaceText = Format("<INPUT TYPE=\"HIDDEN\" \"
            \"NAME=\"CODE\" VALUE=\"%d\">\",
            ARRAYOFCONST((Query1EmpNo->Value)));
}

```

Al otro lado de la línea...

Volvamos la vista atrás por un momento, para evaluar cuánto hemos avanzado. Las técnicas estudiada nos permiten desarrollar aplicaciones o módulos que se ejecutarán en el servidor HTTP, y nos permiten generar dinámicamente el contenido de las páginas HTML. Sin embargo, una vez que dichas páginas llegan al ordenador que las solicita, se convierten en páginas estáticas, de contenido fijo. La única forma de interacción que se le permite al usuario es la navegación por medio de los vínculos incluidos en el documento, y el uso de formularios HTML, que ya hemos visto que son bastante limitados. Puede interesarnos, por ejemplo, realizar algún tipo de validación sobre los datos tecleados en un cuadro de edición antes de enviar su contenido a través de Internet. Estas verificaciones puede, por supuesto, ejecutarlas la propia extensión del servidor, pero el usuario tendría que esperar por la respuesta del servidor para saber que ha tecleado datos no aceptables.

Esta situación ha propiciado la aparición de diversas técnicas aplicables al lado cliente para permitir extensiones al lenguaje HTML básico. Algunas de ellas están basadas en el uso de lenguajes de *script*, como JavaScript y VBScript. En este modelo de documento, se asocian eventos a las diversas etiquetas permitidas por HTML. Cuando se produce un evento, se ejecutan instrucciones programadas con los lenguajes mencionados:

```

<HTML>
<HEAD><TITLE>Demostración sencilla de JavaScript</TITLE></HEAD>
<BODY><H2 onmouseover="MakeBlue();" onmouseout="MakeBlack();" >
Pasa el ratón por encima de este título</H2><HR>

<FORM METHOD="GET" ACTION="www.marteens.com/encuesta.dll/nuevo"
    onsubmit="CheckParams(Nombre);" >
<P>Su nombre: <INPUT NAME="Nombre"></P>
<INPUT TYPE="SUBMIT" VALUE="Enviar">
</FORM>
</BODY>

<SCRIPT LANGUAGE=JavaScript>
function MakeBlue() {
    window.event.srcElement.style.color = "Blue";
}
function MakeBlack() {
    window.event.srcElement.style.color = "Black";
}
function CheckParams(inpField) {
    if (inpField.value == "") {
        alert("Este campo es obligatorio");
    }
}

```

```

        inpField.focus();
        window.event.returnValue = "false";
    }
    else
        window.event.returnValue = "true";
}
</SCRIPT>
</HTML>

```

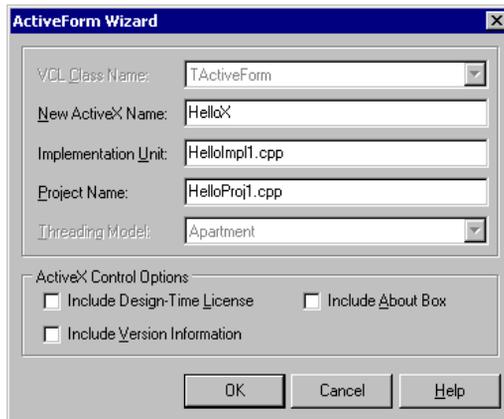
En el fragmento anterior, basado en JavaScript, he señalado con negritas los elementos novedosos. Tenemos un título que cambia de color cuando el ratón pasa sobre él, gracias al tratamiento de los eventos *OnMouseOver* y *OnMouseOut*. Hay también un formulario cuyos datos se validan en respuesta al evento *OnSubmit*. Observe cómo las funciones se describen dentro de una etiqueta <SCRIPT>.

ActiveForms: formularios en la Web

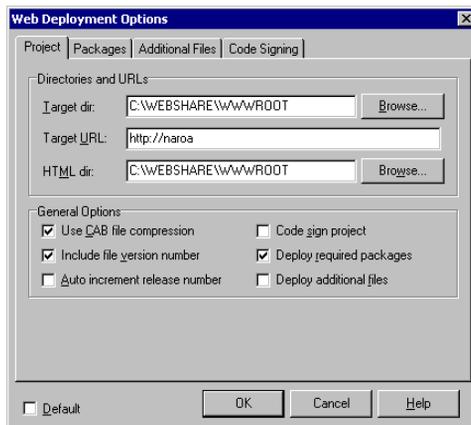
Si hace falta más potencia que ofrecida por estos lenguajes *scripts*, debemos entonces echar mano de algún lenguaje más complejo y potente. Una de las alternativas consiste en el uso de *applets* de Java (no confundir con JavaScript). Estas serían pequeñas aplicaciones diseñadas para ejecutarse en el contexto del navegador de Internet, escritas en Java y traducidas a un código binario interpretable, que es independiente del procesador y del sistema operativo. Otra alternativa casi equivalente consiste en incluir dentro de la página controles ActiveX, que podemos programar con el propio C++ Builder. La desventaja de esta técnica es que los controles ActiveX nos atan a un tipo de procesador (Intel) y a un sistema operativo (Windows).

La forma más sencilla de incluir controles ActiveX en una página Web utilizando C++ Builder consiste en crear un *formulario activo*, o *ActiveForm*, que es sencillamente un control ActiveX con algunas características de un formulario “normal”: puede contener otros componentes VCL y puede actuar como receptor de los eventos por ellos disparados. Para crear un formulario activo debemos ejecutar el asistente *ActiveForm* de la página *ActiveX* del Depósito de Objetos, que se muestra en la página siguiente.

El formulario activo debe crearse dentro del contexto de una biblioteca dinámica ActiveX, para obtener un servidor COM dentro del proceso. De no ser éste el caso, C++ Builder cierra el proyecto activo y crea un nuevo proyecto de este tipo. Observe que el modelo de concurrencia debe ser *Apartment*, pues cada hilo del Internet Explorer de Microsoft se ejecuta en su propio STA.

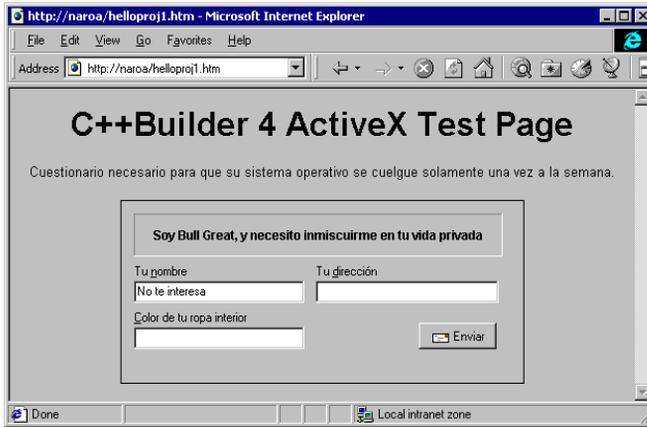


Para el programador, el formulario activo se comporta en tiempo de diseño como cualquier otro formulario. Así que puede colocar cuantos controles desee, asignar propiedades, crear manejadores de eventos, etc. Una vez terminado el diseño del formulario, debe ejecutar el comando de menú *Project | Web Deployment Options*:



La idea es que C++ Builder puede generar automáticamente una página HTML de prueba para nuestra ActiveForm, y mediante el diálogo anterior podemos especificar las opciones de la página y la forma en la que vamos a distribuir el control. En la primera página, por ejemplo, debemos indicar dos directorios y una URL:

Opción	Significado
<i>Target dir</i>	Directorio donde se va a ubicar el control ActiveX compilado y listo para su distribución
<i>Target URL</i>	URL, vista desde un explorador Web, donde residirá el formulario activo
<i>HTML dir</i>	Directorio donde se va a crear la página HTML



En el resto del diálogo podemos indicar si queremos comprimir el formulario (recomendable), si vamos a utilizar paquetes en su distribución, si necesitamos ficheros adicionales, etc. Una vez completado el diálogo, podemos llevar la página HTML y el formulario activo a su ubicación final mediante el comando *Projects | Web deploy*. La imagen anterior muestra un formulario activo cargado localmente con Internet Explorer.

Ahora bien, ¿qué posibilidades reales tenemos de crear aplicaciones serias de bases de datos con esta técnica? En primer lugar, podemos hacer que una aplicación basada en extensiones de servidor utilice formularios activos en el lado cliente como sustitutos de los limitados formularios HTML. Ganaríamos una interfaz de usuario más amigable, pero a costa de perder portabilidad.

La unidad *URLMon* contiene declaraciones de interfaces y rutinas para que nuestro formulario ActiveX pueda comunicarse con el navegador que lo está visualizando. En particular, pueden interesarnos las funciones con el prefijo *Hlink*, como *HlinkGoForward* y *HlinkGoBack*, que nos permiten cambiar la página activa. También podemos utilizar los componentes de la página *Internet* para enviar correo electrónico, o abrir *sockets* que se comuniquen con el servidor HTTP.

La otra oportunidad consiste en situar en el formulario activo componentes de acceso a bases de datos. ¿Quiero decir conjuntos de datos del BDE? ¡No! En tal caso estaríamos complicando aún más las condiciones de configuración en el cliente. De lo que se trata es de utilizar clientes “delgados” como los que desarrollamos con *Midas* en el capítulo 37, para lo cual debemos colocar un servidor *Midas* en una dirección accesible para los usuarios de Internet. El cliente se conectaría al servidor utilizando esa dirección IP con un componente *TSocketConnection*. La principal dificultad tiene que ver con la seguridad: cómo decidir si un usuario tiene derecho o no a acceder al servidor. Existen complicaciones adicionales relacionadas con la configu-

ración del servidor. Muchos servidores de Internet utilizan una barrera de fuego para proteger al verdadero servidor HTTP; la barrera filtra las peticiones de conexión a los servicios IP y deja pasar solamente a los usuarios autorizados al servidor protegido. Si queremos utilizar un servidor Midas desde Internet, hay que convencer al administrador de nuestro dominio para que las peticiones al puerto del servidor Midas (por omisión, el 211) sean también aceptadas.

5

Leftoverture

- Impresión de informes con QuickReport
- Análisis gráfico
- Descenso a los abismos
- Creación de instalaciones
- Ejemplos: libretas de ahorro
- Ejemplos: un servidor de Internet

Parte

Impresión de informes con QuickReport

AUNQUE LA PROPAGANDA COMERCIAL intente convencernos de las bondades del libro electrónico, nuestros ojos siempre agradecerán un buen libro impreso en papel. Del mismo modo, toda aplicación de bases de datos debe permitir imprimir la información con la que trabaja. Sistemas de creación e impresión de informes para C++ Builder hay muchos, quizás demasiados. Inicialmente, junto con Delphi 1 se suministraba un producto de Borland, llamado ReportSmith. Era un generador de informes bastante bueno, con posibilidades de impresión de informes por columnas, *master/detail*, tabulares, gráficos, etc. Sin embargo, no tuvo la acogida deseable por parte de los programadores. ¿La razón?: un *runtime* bastante grande e incómodo que distribuir, demasiado ineficiente en tiempo y espacio (estamos hablando de los tiempos de Delphi 1, sobre Windows 3.1, cuando 16 MB en una máquina era bastante memoria) y, sobre todo, una molesta pantalla de presentación con los créditos de Borland, que aparecía cada vez que se imprimía un informe. Posteriormente se descubrió que era muy fácil ocultar esta pantalla, pero estoy seguro de que ésta fue una de las causas no confesadas que motivaron la aparición de toda una variedad de sistemas de informes alternativos.

La historia del producto

En junio de 1995, un programador noruego llamado Allan Lochert colocó en la Internet un pequeño y eficiente sistema de impresión de informes de libre distribución, al que bautizó QuickReport. Era un producto simple y elegante, basado en el principio “lo pequeño es bello”; el código fuente solamente ocupaba 5000 líneas. El sistema fue creciendo y depurándose, llamando la atención de Borland, que lo incluyó como alternativa a ReportSmith en Delphi 2 y C++ Builder 1. Adicionalmente, era posible comprar aparte la versión profesional de QuickReport, que contenía el código fuente y los componentes para 16 y 32 bits.

Cuando apareció la siguiente versión de QuickReport, éste había sido reprogramado de arriba abajo, cambiando totalmente su apariencia y añadiéndose más potencia. La

filosofía del producto seguía siendo la misma, y era fácil, para alguien familiarizado con la versión anterior, crear un informe con la nueva. También era posible convertir de forma automática un informe de la versión 1 a la 2, pero en ciertos casos especiales, la conversión debía complementarse manualmente.

Por desgracia, junto a la inclusión de nuevas características aparecieron visitantes no deseados: me refiero a *bugs*. Durante el año y poco que duró C++ Builder 3, QuSoft (la empresa responsable de QuickReport) sacó parches designados alfabéticamente, desde la A hasta la K. A veces un parche no sólo corregía errores, sino que introducía otros nuevos. Para colmo de males, Borland se deshizo de ReportSmith, pasando el producto a la compañía Strategic Reporting, aunque continuó incluyendo en la Paleta de Componentes el componente *TReport*, que permite establecer una conexión con el motor de impresión de ReportSmith. Este componente está inicialmente escondido, y hay que utilizar el diálogo de propiedades de la Paleta para mostrarlo.

Ahora con C++ Builder 4, tenemos la versión 3 de QuickReport. Solamente el tiempo y la experiencia nos dirá si es una versión estable y fiable. Esperemos que sí (presunción de inocencia).

En este capítulo estudiaremos solamente las versiones 2 y 3, pues la versión 1 ha quedado obsoleta. Cuando tenga que referirme a alguna versión en específico, utilizaré las siglas QR1, QR2 y QR3 por brevedad. He aquí la Paleta de Componentes de QR3:



La filosofía del producto

QuickReport es un sistema de informes basado en *bandas*. Esto quiere decir que durante el diseño del informe no se ve realmente la apariencia final de la impresión, sino un simple esquema, aunque bastante realista. Tomemos un listado simple de una base de datos: la mayor parte de una página estará ocupada con las líneas procedentes de los registros de la tabla. Pues bien, todas estas líneas tienen la misma función y formato y, en la terminología de QuickReport se dice que proceden de una misma banda: la banda de *detalles*. En ese mismo listado se pueden identificar otras bandas: una correspondiente a la cabecera de páginas, la de pie de páginas, etc. Son estas bandas las que se editan y configuran en QuickReport. Durante la edición, la banda de detalles no se repite, como sucederá durante la impresión. Pero en cualquier momento podemos ver el aspecto final del informe mediante el comando *Preview*.

La otra característica singular es que el proceso de diseño tiene lugar dentro de C++ Builder, utilizando las propias herramientas de diseño del Entorno de Desarrollo. Los componentes de QuickReport se colocan en un formulario, aunque este formulario solamente sirve como contenedor, y nunca es mostrado al usuario de la aplicación. Como consecuencia, todo el motor de impresión reside dentro del mismo espacio de la aplicación; no es un programa externo con carga independiente. Con esto evitamos las demoras relacionadas con la carga en memoria de un programa de impresión externo. Por supuesto, en C++ Builder 3 y 4 el código del motor puede utilizarse desde un paquete; *qrpt30.dpl* ó *qrpt40.bpl*, según la versión.

Otra ventaja de QuickReport es que los datos a imprimir se extraen directamente de conjuntos de datos de C++ Builder. Una tabla que se está visualizando en una ventana de exploración, sobre la cual hemos aplicado filtros y criterios de ordenación, puede también utilizarse de forma directa para la impresión de un informe. En el listado solamente aparecerán las filas aceptadas por el filtro, en el orden especificado para la tabla. El hecho de que los datos salgan directamente de la aplicación implica que no son necesarias conexiones adicionales durante la impresión del informe. Y esto significa muchas veces, en dependencia del servidor, ahorrar en el número de licencias.

Claro está, también existen inconvenientes para este estilo de creación de informes. Ya hemos mencionado el primero: no hay una retroalimentación inmediata de las acciones de edición. El segundo inconveniente es más sutil. Con un sistema de informes independiente se pueden incluir *a posteriori* informes adicionales, que puede diseñar el propio usuario de la aplicación. Esto no puede realizarse directamente con QuickReport, a no ser que montemos un sofisticado mecanismo basado en DLLs o en Automatización OLE (como el desarrollado en el capítulo 36), o un formato de intercambio diseñado desde cero.

Plantillas y expertos para QuickReport

QuickReport trae plantillas de formularios para acelerar la creación de informes, y las podemos encontrar en la página *Forms* del Depósito de Objetos. Las plantillas son tres: una para listados de una sola tabla, una para informes *master/detail*, y otra para la impresión de etiquetas. No voy a entrar en detalles acerca del trabajo con estas plantillas pues, a partir de la explicación que haré de los componentes de impresión, puede deducirse fácilmente qué se puede hacer con ellas.

C++ Builder también ofrece un experto para la generación de listados simples, en la página *Business* del Depósito. Cuando ejecutamos el experto, en la primera página debemos indicar el tipo de informe que queremos generar. En la versión de QuickReport que viene con C++ Builder, solamente tenemos una posibilidad: *List Report*, es decir, un listado simple. En la siguiente pantalla, debemos seleccionar la tabla cuyos

datos vamos a imprimir, ya sea mediante un alias o un directorio, si se trata de una base de datos local. Además, debemos elegir qué campos de la tabla seleccionada queremos mostrar:



El resultado es un formulario con una tabla y con los componentes necesarios de QuickReport. Si quiere visualizar el informe generado, pulse el botón derecho del ratón sobre el componente *TQuickRep* y ejecute el comando *Preview*.

El generador de informes de la versión anterior fallaba constantemente. La secuencia de pantallas era ligeramente diferente a la del experto de C++ Builder 4, y la tabla que creaba la dejaba cerrada, por lo que el comando *Preview* no funcionaba inmediatamente.

El corazón de un informe

Ahora estudiaremos cómo montar manualmente los componentes necesarios para un informe. Para definir un informe, traemos a un formulario vacío un componente *TQuickRep*, que ocupará un área dentro del formulario en representación de una página del listado. Luego hay que asignar la propiedad fundamental e imprescindible, que indica de qué tabla principal se extraen los datos que se van a imprimir: *DataSet*. Esta propiedad, como su nombre indica apunta a un conjunto de datos, no a una fuente de datos. Si se trata de un informe *master/detail*, la tabla que se asigna en *DataSet* es la maestra.

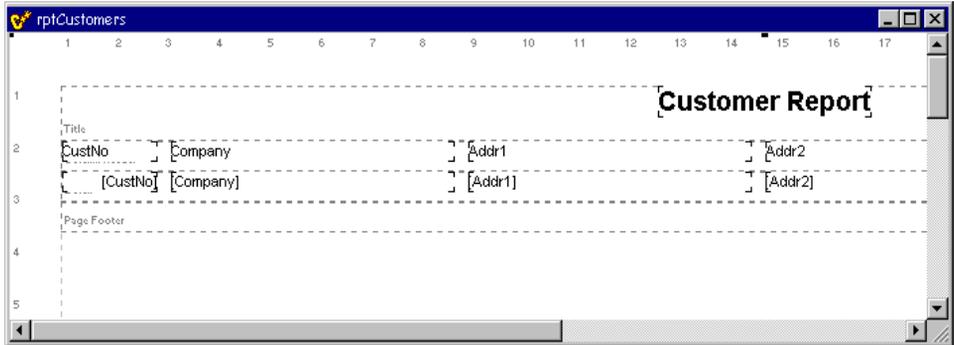
La tarea principal de un componente *TQuickRep* es la de iniciar el proceso de impresión, cuando se le aplica uno de los métodos *Print* ó *Preview*. También podemos utilizar el método *PrintBackground*, que realiza la impresión en un proceso en segundo plano. Por ejemplo, el informe generado en la sección anterior se puede imprimir en respuesta a un comando de menú de la ventana principal utilizando el siguiente código:

```

void __fastcall TForm1::Imprimir1Click(TObject *Sender)
{
    Form2->QuickRep1->Print();
}

```

La propiedad *ReportTitle* del informe se utiliza como título de la ventana de previsualización predefinida.

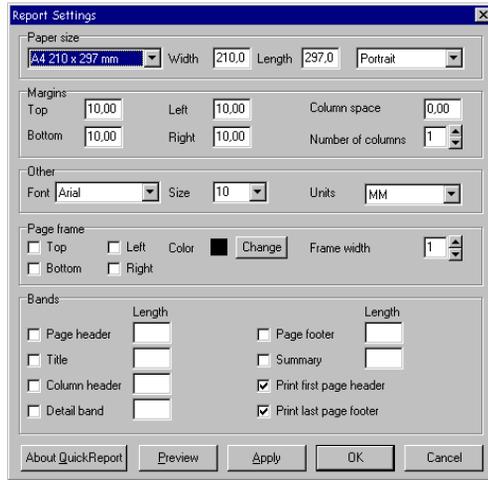


Para obtener la vista preliminar de un informe en tiempo de diseño, utilice el comando *Preview* del menú local del componente. Tenga en cuenta que en tiempo de diseño no se ejecutan los métodos asociados a eventos, así que para ver cualquier opción que haya implementado por código, tendrá ejecutar su aplicación.

La configuración de un informe comienza normalmente por definir las características de la página, que se almacenan dentro de la propiedad *Page*. Esta es una clase con las siguientes propiedades anidadas:

Propiedad	Significado
<i>PaperSize, Length, Width</i>	Tamaño del papel
<i>Orientation</i>	Orientación de la página
<i>LeftMargin, RightMargin</i>	Ancho de los márgenes horizontales
<i>TopMargin, BottomMargin</i>	Ancho de los márgenes verticales
<i>Ruler</i>	Mostrar regla en tiempo de diseño
<i>Columns</i>	Número de columnas
<i>ColumnSpace</i>	Espacio entre columnas

Sin embargo, es más cómodo realizar una doble pulsación sobre el componente *TQuickRep*, para cambiar las propiedades anteriores mediante el siguiente cuadro de edición:



La configuración del tamaño de papel es una de las mayores frustraciones de los programadores de QuickReport, aunque la culpa no es achacable por completo a este producto. En primer lugar, no todos los tamaños de papel son aceptados por todas las impresoras. En segundo lugar, el controlador para Windows de la mayoría de las impresoras no permite definir tamaños personalizados de papel.

Como puede verse en el editor del componente, podemos asignar un tipo de letra por omisión, global a todo el informe. Esta corresponde a la propiedad *Font* del componente *TQuickRep*, y por omisión se emplea la Arial de 10 puntos, que es bastante legible. En el mismo recuadro, a la derecha, se establece en qué unidades se indican las medidas (propiedad *Units*); inicialmente se utilizan milímetros. En el recuadro siguiente se muestran las subpropiedades de la propiedad *Frame*, que sirve para trazar un recuadro alrededor de la página. Pueden seleccionarse las líneas que se van a dibujar, el color, el ancho y su estilo.

Las bandas

Una *banda* es un componente sobre el cual se colocan los componentes “imprimibles”; en este sentido, una banda actúa como un panel. Sobre las bandas se colocan los *componentes de impresión*, en la posición aproximada en la que queremos que aparezcan impresos. Para ayudarnos en esta tarea, el componente *TQuickRep* muestra un par de reglas en los bordes de la página, que se muestran y ocultan mediante el atributo *Ruler* de la propiedad *Page*.

La propiedad más importante de una banda es *BandType*, y estos son sus posibles valores:

Tipo de banda	Objetivo
<i>rbTitle</i>	Se imprime una sola vez, al principio del informe
<i>rbSummary</i>	Se imprime una sola vez, al terminar el informe
<i>rbPageHeader</i>	Se imprime en cada página, en la cabecera
<i>rbPageFooter</i>	Se imprime en cada página, al final de la misma
<i>rbColumnHeader</i>	Si la página se divide en columnas, al comienzo de cada una
<i>rbDetail</i>	Se imprime para cada registro de la tabla principal
<i>rbSubDetail</i>	Se imprime para cada registro de una tabla dependiente
<i>rbGroupHeader</i>	Se imprime cuando se detecta un cambio de grupo
<i>rbGroupFooter</i>	Se imprime cuando termina la impresión de un grupo
<i>rbChild</i>	Banda hija: se imprime siempre después de su banda madre
<i>rbOverlay</i>	Se sobreimprime sobre cada página

El orden de impresión de los diferentes tipos de bandas, para un listado simple, es el siguiente:

<i>rbPageHeader</i> : En todas las páginas
<i>rbTitle</i> : En la primera página
<i>rbColumnHeader</i> : En cada columna, si las hay
<i>rbDetail</i> : Una banda por cada fila de la tabla
<i>rbSummary</i> : Al final del informe
<i>rbPageFooter</i> : Al final de cada página

Las bandas de tipo *rbChild* se imprimen siempre a continuación de la banda madre, pero podemos ejercer más control sobre ellas mediante eventos. En la siguiente sección veremos un ejemplo.

Las bandas pueden añadirse manualmente al informe. Traemos un componente *TQRBand* desde la Paleta de Componentes, e indicamos el tipo de banda en su propiedad *BandType*. Sin embargo, es más fácil indicar las bandas que necesitamos en el Editor de Componente de *TQuickRep*, en el panel inferior, o mediante la propiedad *Bands* del componente. Tenga en cuenta que los componentes *TQRGroup* y *TQRSubDetail*, que estudiaremos más adelante, son componentes visuales y traen incorporadas sus respectivas bandas. En la primera versión de QuickReport, estos componentes venían separados de sus bandas.

La propiedad *Options* del componente *TQuickRep* permite omitir la impresión de la cabecera en la primera página del listado, y la del pie de página en la última. Esto también puede especificarse en el Editor del componente, en el panel inferior, mediante dos casillas de verificación. De este modo, si definimos una banda de título (*rbTitle*) en el informe, se logra el efecto de tener una cabecera de página diferente para la primera página y para las restantes. Sin embargo, las bandas de resumen (*rbSummary*) se imprimen por omisión justo a continuación de la última banda de

detalles. Si queremos que aparezca como si fuera un pie de página, debemos asignar *True* a su propiedad *AlignToBottom*.

El evento *BeforePrint*

Todas las bandas tienen los eventos *BeforePrint* y *AfterPrint*, que se disparan antes y después de su impresión. Podemos utilizar estos eventos para modificar características de la banda, o de los componentes que contienen, en tiempo de ejecución. Por ejemplo, si queremos que las líneas de un listado simple salgan a rayas con colores alternos, como un pijama, podemos crear la siguiente respuesta al evento *BeforePrint* de la banda de detalles:

```
void __fastcall TForm2::DetailBand1BeforePrint(
    TQRCustomBand *Sender, bool &PrintBand)
{
    if (Sender->Color == clWhite)
        Sender->Color = clSilver;
    else
        Sender->Color = clWhite;
}
```



También podemos impedir que se imprima una banda en determinadas circunstancias. Supongamos que la columna *Direccion2* de la tabla *tbClientes* tiene valores no nulos para pocas filas. Nos interesa mostrar esta segunda línea de dirección solamente cuando vaya a contener algún valor. La solución más elegante es colocar el componente de impresión correspondiente a la columna (ver la siguiente sección) en una banda hija de la banda de detalles. Para evitar la impresión de bandas vacías, se crea el siguiente manejador para el evento *BeforePrint* de la nueva banda:

```

void __fastcall TForm2::ChildBand1BeforePrint(TQRCustomBand *Sender,
    bool &PrintBand)
{
    PrintBand = ! tbClientesDireccion2->IsNull;
}

```

Componentes de impresión

Una vez que tenemos bandas, podemos colocar componentes de impresión sobre las mismas. Los componentes de impresión de QuickReport son:

Componente	Imprime...
<i>TQRLabel</i>	Un texto fijo de una línea
<i>TQRMemo</i>	Un texto fijo con varias líneas
<i>TQRRichEdit</i>	Un texto fijo en formato RTF
<i>TQRImage</i>	Una imagen fija, en uno de los formatos de Delphi
<i>TQRShape</i>	Una figura geométrica simple
<i>TQRSysData</i>	Fecha actual, número de página, número de registro, etc.
<i>TQRDBText</i>	Un texto extraído de una columna
<i>TQRDBRichEdit</i>	Un texto RTF extraído de una columna
<i>TQRExpr</i>	Una expresión que puede referirse a columnas
<i>TQRDBImage</i>	Una imagen extraída de una columna

Todos estos objetos, aunque pertenecen a clases diferentes, tienen rasgos comunes. La propiedad *AutoSize*, por ejemplo, controla el área de impresión en la dimensión horizontal. Normalmente, esta propiedad debe valer *False*, para evitar que se superpongan entre sí los diferentes componentes de impresión. En tal caso, es necesario agrandar el componente hasta su área máxima de impresión. La posición del texto a imprimir dentro del área asignada se controla mediante la propiedad *Alignment*: a la derecha, al centro o a la izquierda. Ahora bien, el significado de esta propiedad puede verse afectado por el valor de *AlignToBand*. Cuando *AlignToBand* es *True*, *Alignment* indica en qué posición horizontal, con respecto a la banda, se va a imprimir el componente. Da lo mismo, entonces, la posición en que situemos el componente. Todos los componentes de impresión tienen también la propiedad *AutoStretch* que, cuando es *True*, permite que la impresión del componente continúe en varias líneas cuando no cabe en el área de impresión vertical definida.

El componente más frecuentemente empleado es *TQRDBText*, que imprime el contenido de un campo de un conjunto de datos. Hay que configurar sus propiedades *DataSet* y *DataField*. Tiene la ventaja de que aprovecha automáticamente el formato de visualización del campo asociado, algo que no hace el componente alternativo *TQRExpr*, que veremos a continuación. Este componente, además, es capaz de mostrar el contenido de un campo memo, sin mayores complicaciones.

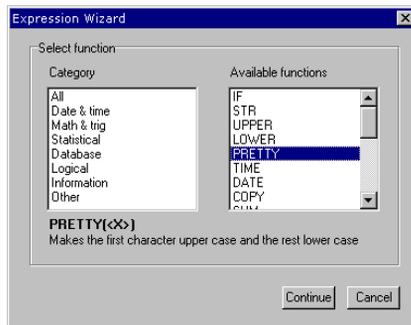
El evaluador de expresiones

Con QuickReport tenemos la posibilidad de imprimir directamente expresiones que utilizan campos de tablas. Muchas veces, podemos utilizar campos calculados con este propósito, por ejemplo, si queremos imprimir el nombre completo de un empleado, teniendo como columnas bases el nombre y los apellidos por separado. Esto puede ser engorroso, sobre todo si la nueva columna es una necesidad exclusiva del informe, pues tenemos que tener cuidado de que el campo no se visualice accidentalmente en otras partes de la aplicación. Pero también cabe utilizar un componente *TQRExpr*, configurando su propiedad *Expression* como sigue:

```
tbEmpleados.LastName + ', ' + tbEmpleados.FirstName
```



Para ayudarnos a teclear las expresiones, QuickReport ofrece un editor especializado para la propiedad *Expression*. El editor es un cuadro de diálogo en el cual podemos formar expresiones a partir de constantes, nombres de campos, operadores y funciones. Una expresión como la anterior se compone muy fácilmente con el editor de expresiones. Como vemos en la imagen anterior, podemos utilizar una serie de funciones en la expresión. El siguiente diálogo aparece cuando pulsamos el botón *Function*:



Una vez elegida una función, pulsamos el botón *Continue*, y aparece el siguiente diálogo, para configurar los argumentos:



Si somos lo suficientemente vagos como para pulsar el botón de la derecha, aparece otra instancia del editor de expresiones, para ayudarnos a componer cada uno de los argumentos de la función.

El componente *TQRExpr* también permite definir estadísticas, si utilizamos las funciones de conjuntos de SQL. Si en una banda *rbSummary* quiere que se imprima la suma total de los salarios de la tabla de empleados, utilice la expresión *Sum(Salary)*. La propiedad *Reset.AfterPrint* indica si se limpia el acumulador del evaluador después de imprimir el componente. No olvide configurar también la propiedad *Master* de la expresión, pues en caso contrario la evaluación no es correcta. En un informe simple, debemos indicar aquí el componente *QuickRep* correspondiente, pero en un informe *master/detail* o que está basado en grupos, hay que indicar el componente *TQRSubDetail* ó *TQRGroup* que pertenece al nivel de anidamiento de la función estadística.

Podemos tener problemas al tratar de imprimir campos de tablas que, aunque son accesibles desde el formulario del informe, no están siendo utilizadas explícitamente por alguno de sus componentes. En tal caso, hay que añadir manualmente la tabla a la lista *AllDataSets* del informe, preferiblemente en el evento *BeforePrint* del mismo.

Utilizando grupos

Al generar un listado, podemos imprimir una banda especial cuando cambia el valor de una columna o expresión en una fila de la tabla base. Por ejemplo, si estamos imprimiendo los datos de clientes y el listado está ordenado por la columna del país, podemos imprimir una línea con el nombre del país cada vez que comienza un grupo

de clientes de un país determinado. De esta forma, puede eliminarse la columna de la banda de detalles, pues ya se imprime en la cabecera de grupo.

Para crear grupos con QuickReport necesitamos el componente *TQRGroup*. Este realiza el cambio controlando el valor que retorna la expresión indicada en la propiedad *Expression*. La siguiente expresión, por ejemplo, provoca que, en un listado de clientes ordenados alfabéticamente, la banda de cabecera de grupo se imprima cada vez que aparezca un cliente con una letra inicial diferente:

```
COPY(tbClientes.Company, 1, 1)
```

Ya hemos mencionado que el componente *TQRGroup* actúa también como banda de cabecera. Los componentes que se deben imprimir en la cabecera del grupo pueden colocarse directamente sobre el grupo. Para indicar la banda del pie de grupo sigue existiendo un propiedad *FooterBand*. Usted trae una banda con sus propiedades predefinidas, y la asigna en esta propiedad. Entonces QuickReport cambia automáticamente el tipo de la banda a *rbGroupFooter*.

Demostraré el uso de grupos con un ejemplo sencillo: quiero un listado de clientes agrupados por totales de ventas: los que han comprado entre 0 y \$25.000, los que han comprado hasta \$50.000, etc. La base del listado es la siguiente consulta, que se coloca en un componente *TQuery*:

```
select Company, sum(ItemsTotal) Total
from Customer, Orders
where Customer.CustNo = Orders.CustNo
group by Company
order by Total desc
```

Es muy importante que la consulta esté ordenada, en este caso por los totales de ventas, en forma descendente. Colocamos un componente *TQuickRep* en un formulario y asignamos el componente *TQuery* que contiene la consulta anterior a su propiedad *DataSet*. Después, con ayuda del editor del informe, añadimos una banda de detalles y una cabecera de página. En la banda de detalles situamos un par de componentes *QRDBText*, asociados respectivamente a cada una de las columnas de la consulta. En este punto, podemos visualizar el resultado de la impresión haciendo clic con el botón derecho del ratón sobre el componente *QuickRep1* y seleccionando el comando *Preview*.

Ahora traemos un componente *TQRGroup* al informe, y editamos su propiedad *Expression*, para asignarle el siguiente texto:

```
INT(Total / 25000)
```

Esto quiere decir que los clientes que hayan comprado \$160.000 y \$140.000 se imprimirán en grupos diferentes, pues la expresión que hemos suministrado devuelve 6 en

el primer caso, y 5 en el segundo. Cuando se inicie un nuevo grupo se imprimirá la banda de cabecera de grupo. Traemos un *TQRExpr* sobre el grupo, y tecleamos la siguiente fórmula en su propiedad *Expression*:

```
'Más de ' + STR(INT(Total / 25000) * 25000)
```

Cada grupo mostrará entonces el criterio de separación adecuado. Para terminar, añada una banda directamente sobre el formulario. Modifique la propiedad *BandType* a *rbGroupFooter*, y *Name* a *PieDeGrupo*. Seleccione el grupo, *QRGroup1*, y asigne *PieDeGrupo* en su propiedad *FooterBand*. Por último, coloque un *TQRExpr* sobre la nueva banda, con la propiedad *Reset.AfterPrint* activa y la siguiente expresión:

```
SUM(Total)
```

De este modo, cuando termine cada grupo se imprimirá el total de todas las compras realizadas por los clientes de ese grupo.

Empresa	Total
<i>Más de 250000</i>	
Sight Diver	\$261.575,80
<i>Total:</i>	\$261.575,80
<i>Más de 175000</i>	
VIP Divers Club	\$193.079,75
American SCUBA Supply	\$183.094,40
<i>Total:</i>	\$376.174,15
<i>Más de 150000</i>	
Blue Sports	\$165.245,45
<i>Total:</i>	\$165.245,45

La versión 3 de QuickReport introduce la propiedad *RepeatOnNewPage*, de tipo *Boolean*. Cuando esta propiedad está activa, las cabeceras de grupo se repiten al inicio de las páginas, si es que el grupo ocupa varias páginas.

Eliminando duplicados

Los grupos de QuickReport, sin embargo, no nos permiten resolver todos los casos de información redundante en un listado. Por ejemplo, tomemos un listado de clientes agrupados por ciudades:

Shangri-La Sports Center	Freeport
Unisco	Freeport
Blue Sports	Giribaldi
Cayman Divers World Unlimited	Grand Cayman
Safari Under the Sea	Grand Cayman

Una forma de mejorar la legibilidad del listado es no repetir el nombre de la ciudad en una línea, si coincide con el nombre de ciudad en la línea anterior:

Shangri-La Sports Center	Freeport
Unisco	
Blue Sports	Giribaldi
Cayman Divers World Unlimited	Grand Cayman
Safari Under the Sea	

Si utilizamos el componente *TQRGroup*, agrupando por ciudad, el nombre de la ciudad no puede aparecer en la misma banda que el nombre de la compañía. Sin embargo, es muy fácil lograr el efecto deseado, manejando el evento *OnPrint* del componente que imprime el nombre de la ciudad. Primero debemos declarar una variable *UltimaCiudad* en la sección **private** del formulario:

```
private:
    AnsiString UltimaCiudad;
    // ...
```

Vamos a inicializar esa variable en el evento *BeforePrint* del informe:

```
void __fastcall TForm2::QuickRep1BeforePrint(
    TCustomQuickRep *Sender, bool &PrintReport)
{
    UltimaCiudad = "";
}
```

Suponiendo que el componente que imprime el nombre de ciudad sea *QRDBText2*, interceptamos su evento *OnPrint*:

```
void __fastcall TForm2::QRExpr2Print(TObject *Sender,
    AnsiString &Value)
{
    if (UltimaCiudad == Value)
        Value = "";
    else
        UltimaCiudad = Value;
}
```

Informes *master/detail*

Existen dos formas de imprimir datos almacenados en tablas que se encuentran en relación *master/detail*. La primera consiste en utilizar una consulta SQL basada en el encuentro natural de las dos tablas, dividiendo el informe en grupos definidos por la clave primaria de la tabla maestra. Por ejemplo, para imprimir un listado de clientes con sus números de pedidos y las fechas de ventas necesitamos la siguiente instrucción:

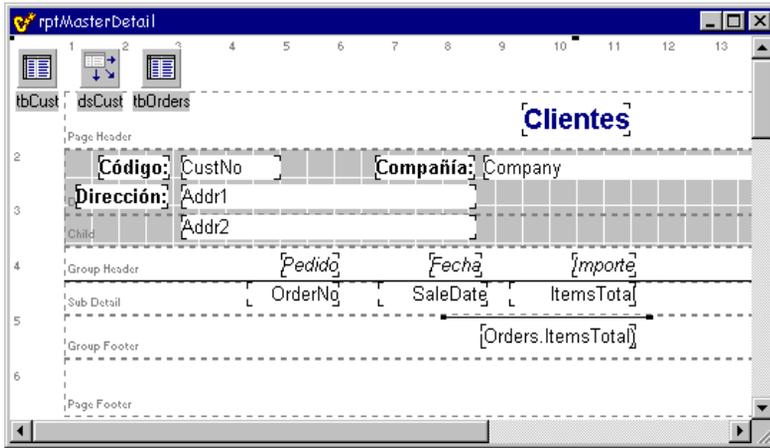
```
select C.CustNo, C.Company, O.OrderNo, O.SaleDate, O.ItemsTotal
from Customer C, Orders O
where C.CustNo = O.CustNo
```

El componente de grupo estaría basado en la columna *CustNo* del resultado. En la cabecera de grupo se imprimirían las columnas *CustNo* y *Company*, mientras que en las filas de detalles se mostrarían *OrderNo*, *SaleDate* y *ItemsTotal*.

La alternativa es emplear el componente *TQRSubDetail* y dejar que QuickReport se ocupe de la gestión de las relaciones entre tablas. Este componente es también una banda, del mismo modo que *TQRGroup*, y sus propiedades principales son:

Propiedad	Significado
<i>Master</i>	Apunta a un <i>TQuickReport</i> o a otro <i>TQRSubDetail</i>
<i>DataSet</i>	El conjunto de datos de detalles
<i>Bands</i>	Permite crear rápidamente la cabecera y el pie
<i>HeaderBand</i>	Una banda de tipo <i>rbGroupHeader</i>
<i>FooterBand</i>	Una banda de tipo <i>rbGroupFooter</i>

Tomemos como ejemplo el informe descrito anteriormente, acerca de clientes y pedidos, y supongamos que esta vez tenemos dos tablas, *tbClientes* y *tbPedidos*, enlazadas en relación *master/detail*. Para crear el informe correspondiente, añada un componente *TQuickRep* a un formulario vacío y cambie su propiedad *DataSet* a *tbClientes*. Cree ahora una banda de detalles, *rbDetail*, que es donde colocaremos los componentes de impresión correspondientes a la tabla de clientes. Ahora coloque un *TQRSubDetail*, con su propiedad *DataSet* igual a *tbPedidos*. Encima de éste se colocarán los componentes de impresión de los pedidos. Ajuste la propiedad *Master* del enlace de detalles a *QuickRep1*; la propiedad *Master* puede apuntar también a otro componente *TQRSubDetail*, permitiendo la impresión de informes con varios niveles de detalles.



Para crear las bandas de cabecera y pie de grupo, expanda la propiedad *Bands* del *TQRSubDetail*. La clase a la que pertenece *Bands* contiene las propiedades *HasHeader* y *HasFooter*. Basta con asignar *True* a las dos para que se añadan y configuren automáticamente las dos bandas indicadas. De este modo, queda configurado el esqueleto del informe. Solamente queda colocar sobre las bandas los componentes de impresión, de tipo *TQRDBText*, que queremos que aparezcan en el informe.

Por supuesto, es incómodo tener que configurar los componentes de la manera explicada cada vez que necesitamos un informe con varias tablas. Por ese motivo, todas las versiones de QuickReport incluyen una plantilla en la cual ya están incluidas las bandas necesarias para este tipo de informe. Solamente necesitamos añadir sobre las mismas los componentes de impresión correspondientes.

Informes compuestos

QR2 introduce el componente *TQRCompositeReport*, que sirve para imprimir consecutivamente varios informes. Este componente tiene los métodos *Print*, *PrintBackground* y *Preview*, al igual que un informe normal, para imprimir o visualizar el resultado de su ejecución. Pongamos por caso que queremos un listado de una tabla con muchas columnas, de modo que es imposible colocar todas las columnas en una misma línea. Por lo tanto, diseñamos varios informes con subconjuntos de las columnas: en el primer informe se listan las siete primeras columnas, en el segundo las nueve siguientes, etc. A la hora de imprimir estos informes queremos que el usuario de la aplicación utilice un solo comando. Una solución es agrupar los informes individuales en un único informe, y controlar la impresión desde éste. Podemos traer entonces un componente *QRCompositeReport1* e interceptar su evento *OnAddReports*:

```

void __fastcall TForm1::QRCompositeReport1AddReports (
    TObject* Sender)
{
    QRCompositeReport1->Reports->Add(Form2->QuickReport1);
    QRCompositeReport1->Reports->Add(Form3->QuickReport1);
    QRCompositeReport1->Reports->Add(Form4->QuickReport1);
}

```

Normalmente, los informes se imprimen uno a continuación del otro. La documentación indica que basta con asignar *True* a la propiedad *ForceNewPage* de la banda de título de un informe para que este comience su impresión en una nueva página. Sin embargo, esto no funciona. Lo que sí puede hacerse es crear un manejador para el evento *BeforePrint* del informe:

```

void __fastcall TInforme3::TitleBand1BeforePrint (
    TQRCustomBand *Sender, bool &PrintBand)
{
    QuickRep1->NewPage();
}

```

Al parecer, QuickReport ignora la propiedad *ForceNewPage* cuando se debe aplicar en la primera página de un informe.

Cuando se utilizan informes compuestos, todos los informes individuales deben tener el mismo tamaño de papel y la misma orientación.

Previsualización a la medida

Las versiones anteriores de QuickReport solamente permitían la vista preliminar en forma no modal. En la versión que acompaña a C++ Builder 4 existen tres métodos para este propósito:

Método	Resultados
<i>Preview</i>	Vista modal; impresión en el hilo principal del programa.
<i>PreviewModal</i>	Vista modal; impresión en hilo paralelo.
<i>PreviewModeless</i>	Vista no modal; impresión en hilo paralelo.

Me estoy ajustando a la escasa documentación de QuickReport para describir las diferencias entre los tres métodos anteriores. Algunos programadores han encontrado pérdidas de memoria cuando utilizan métodos de previsualización distintos de *PreviewModal*.

El programador puede también crear un diálogo o ventana de previsualización a la medida. La base de esta técnica es el componente *TQRPreview*. El diseño de una ventana de previsualización a la medida comienza por crear un formulario, al cual

llamaremos *Visualizador*, e incluir en su interior un componente *TQRPreview*. A este formulario básico pueden añadirse entonces componentes para controlar el grado de acercamiento, el número de página, la impresión, etc. Digamos, para precisar, que el formulario que contiene el informe se llama *Informe*. Para garantizar la liberación de los recursos asignados al formulario de previsualización, debemos interceptar su evento *OnClose*:

```
void __fastcall TVisualizador::FormClose(TObject *Sender,
TCloseAction &Action)
{
    QRPreview1->QRPrinter = NULL;
    Action = caFree;
}
```

Después hay que regresar al formulario del informe para interceptar el evento *OnPreview* del componente *QuickRep1* de la siguiente manera:

```
void __fastcall TInforme::QuickRep1Preview(TObject *Sender)
{
    Visualizador = new TVisualizador(NULL);
    Visualizador->QRPreview1->QRPrinter = (TQRPrinter*) Sender;
    Visualizador->ShowModal();
    delete Visualizador;
    // O también: Visualizador->Show(), sin destruir el objeto,
    // si queremos una previsualización no modal.
}
```

El parámetro *Sender* del evento apunta al objeto de impresora que se va a utilizar con el informe. Recuerde que *QuickReport* permite la impresión en paralelo, por lo cual cada informe define un objeto de tipo *TQRPrinter*, diferente del objeto *QRPrinter* global.

Si queremos navegar por la páginas de la vista preliminar, podemos traer botones o comandos de menú al formulario y crear un manejador de eventos como el siguiente:

```
void __fastcall TVisualizador::Navigate(TObject *Sender)
{
    // Utilizaré Tag para distinguir entre acciones
    switch (((TComponent*)Sender)->Tag)
    {
        case 0:
            QRPreview1->PageNumber = 1;
            break;
        case 1:
            QRPreview1->PageNumber -= 1;
            break;
        case 2:
            QRPreview1->PageNumber += 1;
            break;
    }
}
```

```

        case 3:
            QRPreview1->PageNumber = QRPreview1->QRPrinter->PageCount;
            break;
        }
    }
}

```

Para imprimir hace falta el siguiente método:

```

void __fastcall TVisualizador::Navigate(TObject *Sender)
{
    QRPreview1->QRPrinter->Print();
}

```

Existen innumerables variaciones sobre este tema. Si usted va a imprimir un documento con un procesador de texto, puede optar por ver una presentación preliminar y luego imprimir. Pero también puede entrar a saco en el diálogo de impresión, elegir un rango de páginas e imprimir directamente. Para obtener este comportamiento de QuickReport, necesitamos algo parecido a esto:

```

void __fastcall TForm1::Button1Click(TObject *Sender)
{
    Form2->QuickRep1->PrinterSetup();
    if (Form2->QuickRep1->Tag == 0)
        Form2->QuickRep1->Print();
}

```

Observe que el método *PrinterSetup* del informe es un procedimiento, no una función. Para saber si ha terminado correctamente, o si el usuario ha cancelado el diálogo, hay que revisar el contenido de la propiedad *Tag* del propio informe; si ésta vale cero, todo ha ido bien. Sinceramente, cuando veo cochinas como éstas en los productos de mi fabricante de software favorito, que además no están correctamente documentadas, siento vergüenza ajena.

Listados al vuelo

En la unidad *QRExtra* de QuickReport se ofrecen clases y funciones que permiten generar informes en tiempo de ejecución. Aquí solamente presentaré la técnica más sencilla, basada en el uso de la función *QRCreateList*:

```

void __fastcall QRCreateList(TCustomQuickRep* &aReport,
    TComponent* aOwner, TDataSet* aDataSet, AnsiString aTitle,
    TStringList* aFieldList);

```

En el primer parámetro debemos pasar una variable del tipo *TCustomQuickRep*, el ancestro de *TQuickRep*. Si inicializamos la variable con el puntero *NULL*, la función crea el objeto. Pero si pasamos un objeto creado, éste se aprovecha. *AOwner* corresponde al propietario del informe, *aDataSet* es el conjunto de datos en que se basará el listado, y *aTitle* será el título del mismo. Por último, si pasamos una lista de nombres

de campos en *AFieldList*, podremos indicar qué campos deseamos que aparezcan en el listado. En caso contrario, se utilizarán todos. Este será el aspecto de la aplicación que crearemos:



Ya hemos visto, en el capítulo sobre bases de datos y sesiones, cómo extraer información sobre los alias disponibles, las tablas que existen dentro de los mismos, y los campos que contiene cada tabla. De todos modos, mostraré el código asociado al mantenimiento de esta información. Debo advertir que he utilizado un objeto persistente *TTable* como ayuda:

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    Session->GetAliasNames(ComboBox1->Items);
    ComboBox1->ItemIndex = 0;
    ComboBox1Change(NULL);
}

void __fastcall TForm1::ComboBox1Change(TObject *Sender)
{
    Session->GetTableNames(ComboBox1->Text, "", True, False,
        ComboBox2->Items);
    ComboBox2->ItemIndex = 0;
    ComboBox2Change(NULL);
}

void __fastcall TForm1::ComboBox2Change(TObject *Sender)
{
    CheckListBox1->Clear();
    Table1->DatabaseName = ComboBox1->Text;
    Table1->TableName = ComboBox2->Text;
    Table1->FieldDefs->Update();
    for (int i = 0; i < Table1->FieldDefs->Count; i++)
    {
        CheckListBox1->Items->Add(Table1->FieldDefs->Items[i]->Name);
        CheckListBox1->Checked[i] = True;
    }
}
```

La parte principal de la aplicación es la respuesta al botón de impresión, que mostramos a continuación:

```

void __fastcall TForm1::Button1Click(TObject *Sender)
{
    TCustomQuickRep *MyRep = NULL;
    Table1->Open();
    try
    {
        std::auto_ptr<TStringList> Fields(new TStringList);
        for (int i = 0; i < CheckListBox1->Items->Count; i++)
            if (CheckListBox1->Checked[i])
                Fields->Add(CheckListBox1->Items->Strings[i]);
        if (Fields->Count > 0)
        {
            QRCreateList(MyRep, this, Table1, "", Fields.get());
            try { MyRep->Preview(); }
            __finally { delete MyRep; }
        }
    }
    __finally
    {
        Table1->Close();
    }
}

```

Entre las posibles mejoras que admite el algoritmo están permitir la configuración del tipo de letra, el poder especificar un título (aunque solamente lo utilizará la vista preliminar), y descartar automáticamente los campos gráficos, que esta función no maneja correctamente.

Enviando códigos binarios a una impresora

A veces nuestras principales ventajas se convierten en nuestras limitaciones esenciales. En el caso de QuickReport, se trata del hecho de que toda la impresión se realice a través del controlador para Windows de la impresora que tengamos conectada. Un controlador de impresora en Windows nos permite, básicamente, simular que dibujamos sobre la página activa del dispositivo. Por ejemplo, si tenemos que imprimir un círculo utilizamos la misma rutina de Windows que dibuja un círculo en la pantalla del monitor, y nos evitamos tener que enviar códigos binarios para mover el cabezal de la impresora y manchar cuidadosamente cada píxel de los que conforman la imagen. Y esto es una bendición.

Ahora bien, hay casos en los que nos interesa manejar directamente la impresión. Tal es la situación con determinados informes que requieren impresoras matriciales, o con dispositivos de impresión muy especiales como ciertas impresoras de etiquetas. Aún en el caso en que el fabricante proporcione un controlador para Windows, el rendimiento del mismo será generalmente pobre, al tratar de compaginar el modo gráfico que utiliza Windows con las limitaciones físicas del aparato.

Para estos casos especiales, he decidido incluir en este capítulo una función que permite enviar un *buffer* con códigos binarios directamente a la impresora, haciendo caso

omiso de la interfaz de alto nivel del controlador. La función realiza llamadas al API de Windows, y utiliza la siguiente función auxiliar para transformar valores de retorno de error en nuestras queridas excepciones:

```
void __fastcall PrinterError()
{
    throw Exception("Operación inválida con la impresora");
}
```

No puedo entrar en explicaciones sobre cada una de las funciones utilizadas aquí, pues esto sobrepasa los objetivos del presente libro. He aquí el código:

```
void __fastcall Imprimir(AnsiString PrinterName,
    LPVOID Data, DWORD Count)
{
    // Estructura con la información sobre el documento
    struct {
        char *DocName, *OutputFile, *DataType;
    } DocInfo = {"Document", 0, "RAW"};

    // Obtener un handle de impresora
    HANDLE hPrinter;
    if (! OpenPrinter(PrinterName.c_str(), &hPrinter, 0))
        PrinterError();
    try {
        // Informar al spooler del comienzo de impresión
        if (StartDocPrinter(hPrinter, 1, LPBYTE(&DocInfo)) == 0)
            PrinterError();
        try {
            // Iniciar una página
            if (! StartPagePrinter(hPrinter)) PrinterError();
            try {
                // Enviar los datos directamente
                DWORD Bytes;
                if (! WritePrinter(hPrinter, Data, Count, &Bytes) ||
                    Bytes != Count) PrinterError();
            }
            __finally {
                // Terminar la página
                if (! EndPagePrinter(hPrinter)) PrinterError();
            }
        }
        __finally {
            // Informar al spooler del fin de impresión
            if (! EndDocPrinter(hPrinter)) PrinterError();
        }
    }
    __finally {
        // Devolver el handle de impresora
        ClosePrinter(hPrinter);
    }
}
```

Esta otra versión de *Imprimir* ha sido adaptada para que imprima una secuencia de caracteres y códigos contenidos en una cadena en la impresora:

```
void __fastcall Imprimir(AnsiString PrinterName, AnsiString Data)
{
    Imprimir(PrinterName, LPVOID(Data.c_str()), Data.Length());
}
```

Por ejemplo:

```
Imprimir("Priscilla", "Hola, mundo...\x0D\x0A"
    "...adiós, mundo cruel\x0C");
```

Puede también generar un documento a imprimir enviando primero los códigos necesarios a un objeto *TMemoryStream*, y pasando posteriormente su propiedad *Memory* a la función *Imprimir*. *TMemoryStream* fue estudiado en el capítulo sobre tipos de datos.

Análisis gráfico

ENTRE LOS COMPONENTES INTRODUCIDOS por la versión 3 de la VCL, destacan los que se encuentran en la página *Decision Cube*, que nos permiten calcular y mostrar gráficos y rejillas de decisión. Con los componentes de esta página podemos visualizar en pantalla informes al estilo de *tablas cruzadas (crosstabs)*. En este tipo de informes se muestran estadísticas acerca de ciertos datos: totales, cantidades y promedios, con la particularidad de que el criterio de agregación puede ser multidimensional. Por ejemplo, nos interesa saber los totales de ventas por delegaciones, pero a la vez queremos subdividir estos totales por meses o trimestres. Además, queremos ocultar o expandir dinámicamente las dimensiones de análisis, que los resultados se muestren en rejillas, o en gráficos de barras. Todo esto es tarea de *Decision Cube*. Y ya que estamos hablando de gráficos, explicaremos también como aprovechar los componentes *TChart* y *TDBChart*, que permiten mostrar series de datos generadas mediante código o provenientes de tablas y consultas.

Para ilustrar el empleo de *Decision Cube* y del componente *TDBChart* utilizaremos la base de datos correspondiente a una aplicación que desarrollaremos en el capítulo 44, que tratará sobre la gestión de libretas de ahorro. Para mayor facilidad, he incluido una versión en formato Paradox de la misma en el directorio del CD-ROM correspondiente a este capítulo.

Gráficos y biorritmos

La civilización occidental tiende a infravalorar la importancia de los ritmos en nuestra vida. Existen, sin embargo, teorías cognitivas que asocian la génesis de la conciencia con asociaciones rítmicas efectuadas por nuestros antepasados. Muchos mitos que perviven entre nosotros reconocen de un modo u otro este vínculo. Y uno de los mitos modernos relacionados con los ritmos es la “teoría” de los biorritmos: la suposición de que nuestro estado físico, emocional e intelectual es afectado por ciclos de 23, 28 y 33 días respectivamente. Se conjetura que estos ciclos arrancan a partir de la fecha de nacimiento de la persona, de modo que para saber el estado de los mismos para un día determinado basta con calcular el total de días transcurridos desde entonces y realizar la división entera con resto. Al resultado, después de una transfor-

mación lineal sencilla, se le aplica la función seno y, ¡ya hay pronóstico meteorológico!

No voy a describir totalmente el proceso de desarrollo de una aplicación que calcule e imprima biorritmos. Mi interés es mostrar cómo puede utilizarse el componente *TChart* para este propósito. Este componente se encuentra en la página *Additional* de la Paleta de Componentes. Pero si exploramos un poco la Paleta, encontraremos también los componentes *TDBCchart* y *TQRDBCchart*. En principio, *TChart* y *TDBCchart* tienen casi la misma funcionalidad, pero el segundo puede ser llenado a partir de un conjunto de datos, especificando determinados campos del mismo. El conjunto de datos puede ser indistintamente una tabla, una consulta o el componente derivado de *TDataSet* que se le ocurra. Por su parte, *TQRDBCchart* puede incluirse dentro de un informe generado con QuickReport.

Un gráfico debe mostrar valores de una colección de datos simples o puntuales. La colección de datos de un gráfico de tarta, por ejemplo, debe contener pares del tipo:

(etiqueta, proporción)

La proporción determina el ángulo que ocupa cada trozo de la tarta, y la etiqueta sirve ... pues para eso ... para etiquetar el trozo. En cambio, un gráfico lineal puede contener tripletas en vez de pares:

(etiqueta, valor X, valor Y)

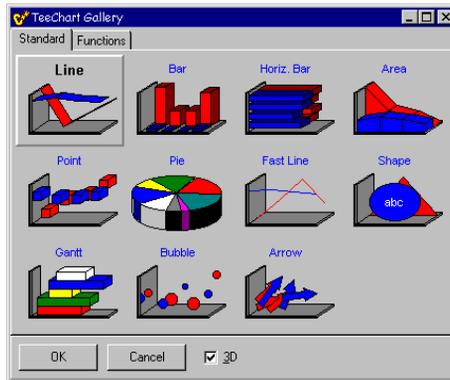
Ahora se ha incluido un valor X para calcular distancias en el eje horizontal. Si un gráfico lineal contiene tres puntos, uno para enero, uno para febrero y otro para diciembre, se espera que los puntos de enero y febrero estén más alejados del punto correspondiente a diciembre.

A estas colecciones de puntos se les denomina *series*. Un componente *TChart* contiene una o más series, que son objetos derivados de la clase *TSeries*. Esto quiere decir que puede mostrar simultáneamente más de un gráfico, en el sentido usual de la palabra, lo cual puede valer para realizar comparaciones. He dicho antes que los tipos de series concretas se derivan de la clase abstracta *TSeries*, y es que la estructura de una serie correspondiente a un gráfico lineal es diferente a la de una serie que contenga los datos de un gráfico de tarta.

Las series deben ser creadas por el programador, casi siempre en tiempo de diseño. Para ilustrar la creación, traiga a un formulario vacío un componente *TChart*, y pulse dos veces sobre el mismo, para invocar a su editor predefinido. Debe aparecer el siguiente cuadro de diálogo:



A continuación pulse el botón *Add*, para que aparezca la galería de estilos disponible. Podemos optar por series simples, o por funciones, que se basan en datos de otras series para crear series calculadas. Escogeremos un gráfico lineal, por simplicidad y conveniencia.



Cuando creamos una serie para un gráfico, estamos creando explícitamente un componente con nombre y una variable asociada dentro del formulario. Por omisión, la serie creada se llamará *Series1*. Repita dos veces más la operación anterior para tener también las series *Series2* y *Series3*. Las tres variables apuntan a objetos de la clase *TLineSeries*, que contiene el siguiente método:

```
int __fastcall Add(const double YValue, const AnsiString ALabel,
    TColor AColor);
```

Utilizaremos este método, en vez de *AddXY*, porque nuestros puntos irán espaciados uniformemente. En el tercer parámetro podemos utilizar la constante especial de color *cTeeColor*, para que el componente decida qué color utilizar.

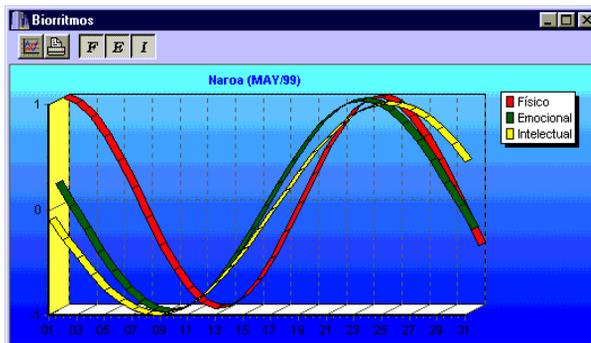
Ahora hay que añadir al programa algún mecanismo para que el usuario pueda teclear una fecha de nacimiento, decir a partir de qué fecha desea el gráfico de los bio-

ritmos, y el número de meses que debe abarcar. Todo esto lo dejo en sus manos expertas. Me limitaré a mostrar la función que rellena el gráfico a partir de estos datos:

```
void __fastcall TwndMain::FillChart(AnsiString Nombre,
    TDateTime Nacimiento, TDateTime PrimerDia, int Meses)
{
    unsigned short Y, M, D;
    double Pi2 = 2 * PI();

    if (Nombre == "")
        Nombre = "BIORRITMOS";
    // Calcular el último día
    PrimerDia.DecodeDate(&Y, &M, &D);
    while (Meses > 0)
    {
        --Meses; ++M;
        if (M > 12)
            M = 1, ++Y;
    }
    TDateTime UltimoDia = TDateTime(Y, M, 1) - 1;

    AnsiString MS1 = FormatDateTime("mmm/yy", PrimerDia).UpperCase();
    AnsiString MS2 = FormatDateTime("mmm/yy", UltimoDia).UpperCase();
    if (MS1 != MS2)
        MS1 = MS1 + "-" + MS2;
    Chart1->Title->Text->Text = Nombre + " (" + MS1 + ")";
    Series1->Clear();
    Series2->Clear();
    Series3->Clear();
    for (int i = 1; i <= int(UltimoDia - PrimerDia) + 1; i++)
    {
        int Dias = int(PrimerDia - Nacimiento);
        AnsiString Etiqueta := FormatDateTime("dd", PrimerDia);
        Series1->Add(sin((Dias % 23) * Pi2 / 23),
            Etiqueta, clTeeColor);
        Series2->Add(sin((Dias % 28) * Pi2 / 28),
            Etiqueta, clTeeColor);
        Series3->Add(sin((Dias % 33) * Pi2 / 33),
            Etiqueta, clTeeColor);
        PrimerDia += 1;
    }
}
```



Como el lector puede observar, he incluido tres botones para que el usuario pueda “apagar” selectivamente la visualización de alguna de las series, mediante la propiedad *Active* de las mismas. La propiedad *Tag* de los tres botones han sido inicializadas a 0, 1 y 2 respectivamente. De este modo, la respuesta al evento *OnClick* puede ser compartida, aprovechando la presencia de la propiedad vectorial *Series* en el gráfico:

```
void __fastcall TwndMain::SwitchSeries(TObject *Sender)
{
    TToolButton &button = dynamic_cast<TToolButton&>(Sender);
    Chart1->Series[button.Tag]->Active = button.Down;
}
```

Vuelvo a advertirle: nunca utilice los biorritmos para intentar predecir su futuro, pues es una verdadera superstición. No es lo mismo, por ejemplo, que mirar en una bola de cristal o echar el tarot. Si el lector lo desea, puede llamarme al 906-999-999, y por una módica suma le diré lo que le tiene reservado el destino.

El componente *TDBChart*

Abra la aplicación de las libretas de banco, y añadimos una nueva ventana a la misma. Mediante el comando de menú *File | Use unit* hacemos que utilice la unidad del módulo de datos. Después colocamos sobre la misma un componente *TPageControl* con tres páginas:

- Evolución del saldo.
- Rejilla de análisis.
- Gráfico de análisis.

En la primera página situaremos un gráfico de línea para mostrar la curva del saldo respecto al tiempo, con el propósito de deprimir al usuario de la aplicación. En la segunda y tercera página desglosaremos los ingresos y extracciones con respecto al concepto de la operación y el mes en que se realizó; en una de ellas utilizaremos una rejilla, mientras que en la segunda mostraremos un gráfico de barras basado en dos series.

Comenzaremos con la curva del saldo. Necesitamos saber qué saldo teníamos en cada fecha; la serie de pares fecha/saldo debe estar ordenada en orden ascendente de las fechas. Aunque podemos usar una tabla ordenada mediante un índice, para mayor generalidad utilizaremos una consulta. Creamos entonces un módulo de datos para la aplicación, y le añadimos un componente *TQuery*. Dentro de su propiedad SQL tecleamos la siguiente instrucción:

```

select Fecha, Saldo
from Apuntes
where Libreta = :Codigo
order by Fecha

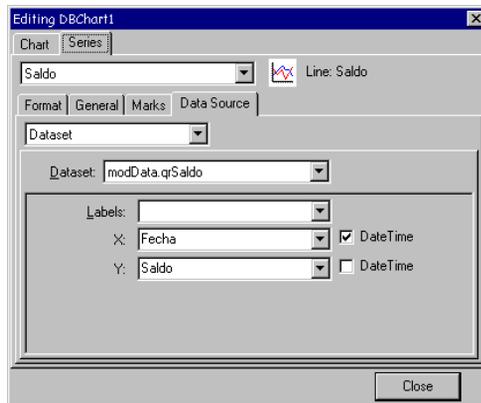
```

La cláusula **where** especifica que solamente nos interesan los apuntes correspondientes a la libreta activa en la aplicación, por lo cual debemos asignar la fuente de datos asociada a la tabla de libretas, *dsLib*, en la propiedad *DataSource* de la consulta. Recuerde que en este caso no debe asignarse un tipo al parámetro *Codigo*, pues corresponde a la columna homónima de la tabla de libretas.

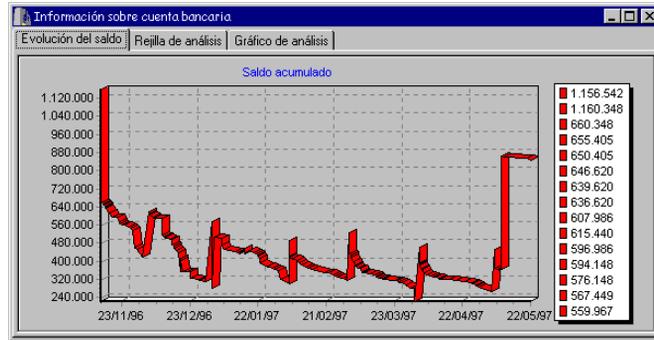
Ahora vamos a la primera página del formulario principal, seleccionamos la página *Data Controls* de la Paleta de Componentes, y traemos un *TDBChart*, cambiándole su alineación *Align* a *alClient*. ¿Por qué *TDBChart*? Porque en este caso, los datos pueden obtenerse directamente de la base de datos.

El próximo paso es crear una serie lineal. Recuerde que basta con realizar un doble clic sobre el componente *TDBChart*. Pulsando después el botón *Add* podemos seleccionar un tipo de serie para añadir: elija nuevamente una serie lineal. Para colocar más gráficos en el mismo formulario y evitar un conflicto que el autor de *TChart* podía haber previsto, cambiaremos el nombre del componente a *Saldos*. Podemos hacerlo seleccionando el componente *Series1* en el Inspector de Objetos, y modificando la propiedad *Name*.

La fuente de datos de la serie se configura esta vez en la página *Series | Data Source*. Tenemos que indicar que alimentaremos a la serie desde un conjunto de datos (en el combo superior), el nombre del conjunto de datos (la consulta que hemos definido antes), y qué columnas elegiremos para los ejes X e Y. En este caso, a diferencia de lo que sucedió con los biorritmos, nos interesa espaciar proporcionalmente los valores del eje horizontal, de acuerdo a la fecha de la operación:



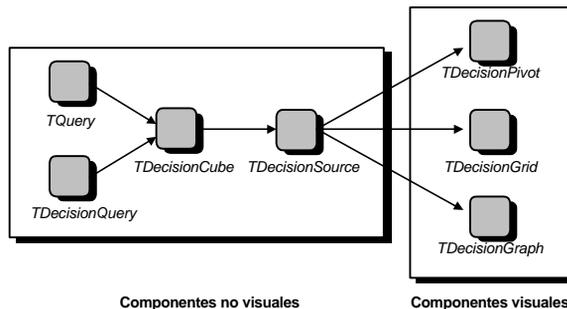
A continuación, podemos configurar detalles del gráfico, como el color del fondo, el título, etc. Puede también intentar añadir una serie que represente el saldo promedio, utilizando la página *Functions* de la galería de series. El gráfico final puede apreciarse en la siguiente figura:



Componentes no visuales de *Decision Cube*

Para preparar los gráficos y rejillas que utilizaremos con *Decision Cube*, necesitamos unos cuantos componentes no visuales, que colocaremos en el módulo de datos. El origen de los datos a visualizar puede ser indistintamente un componente *TQuery* o un *TDecisionQuery*; la diferencia entre ambos es la presencia de un editor de componente en *TDecisionQuery* para ayudar en la generación de la consulta. La consulta se conecta a un componente *TDecisionCube*, que es el que agrupa la información en forma matricial para su uso posterior por otros componentes. El cubo de decisión se conecta a su vez a *TDecisionSource*, que sirve como fuente de datos a los componentes visuales finales: *TDecisionGrid*, para mostrar los datos en formato de rejilla, *TDecisionGraph*, un derivado de los gráficos TeeChart, y *TDecisionPivot*, para manejar dinámicamente las dimensiones del gráfico y la rejilla.

El siguiente diagrama muestra cómo se conectan los distintos componentes de esta página:



Utilizaremos, para simplificar la exposición, una consulta común y corriente como el conjunto de datos que alimenta toda la conexión anterior; la situaremos, como es lógico, en el módulo de datos de la aplicación. Este es el contenido de la instrucción SQL que debemos teclear dentro de un componente *TQuery*:

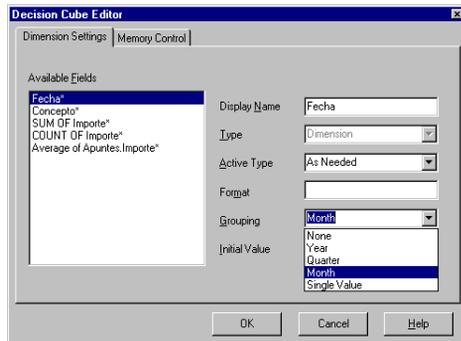
```

select A.Fecha, C.Concepto, sum(A.Importe), count(A.Importe)
from Apuntes A, Conceptos C
where A.Concepto = C.Codigo and
        A.Libreta = :Codigo
group by A.Fecha, C.Concepto
    
```

Nuevamente, hemos restringido el resultado a los apuntes correspondientes a la libreta actual. Recuerde asignar *dsLib* a la propiedad *DataSource* de esta consulta.

Queremos mostrar la suma de los importes de las operaciones, el promedio y la cantidad de las mismas, desglosadas según la fecha y el concepto; del concepto queremos la descripción literal, no su código. Para cada dimensión, se incluye directamente en la consulta la columna correspondiente, y se menciona dentro de una cláusula **group by**. Por su parte, los datos a mostrar se asocian a expresiones que hacen uso de funciones de conjuntos. En el ejemplo hemos incluido la suma y la cantidad del importe; no es necesario incluir el promedio mediante la función **avg**, pues Decision Cube puede deducirlo a partir de los otros dos valores.

Una vez que está configurada la consulta, traemos un componente *TDecisionCube*, lo conectamos a la consulta anterior mediante su propiedad *DataSet*, y realizamos un doble clic sobre el componente para su configuración. Necesitamos cambiar el título de las dimensiones y de las estadísticas, cambiando títulos como “*SUM OF Importe*” a “*Importe total*”. Es muy importante configurar las dimensiones de tipo fecha, indicando el criterio de agrupamiento. En este ejemplo hemos cambiando la propiedad *Grouping* de las fechas de los apuntes a *Month*, para agrupar por meses; también pueden agruparse por días, años y trimestres. Para terminar, colocamos un componente *TDecisionSource* dentro del módulo de datos, y asignamos *DecisionCube1* a su propiedad *DecisionCube*.



Rejillas y gráficos de decisión

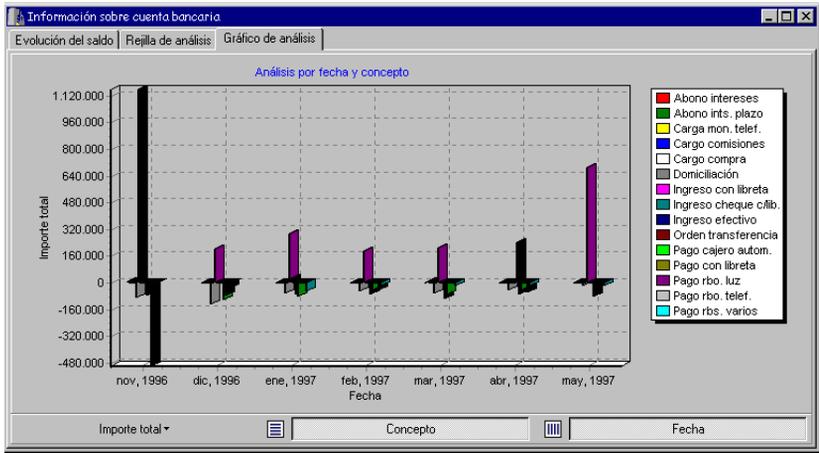
Es el momento de configurar los componentes visuales. Vamos a la segunda página del formulario principal, y añadimos un componente *TDecisionGrid* en la misma, cambiando su alineación a *alClient*, para que ocupe toda el área interior de la página. Este es el aspecto de la rejilla:

Información sobre cuenta bancaria		Evolución del saldo						
		Rejilla de análisis						
		Gráfico de análisis						
Concepto	Fecha	nov, 1996	dic, 1996	ene, 1997	feb, 1997	mar, 1997	abr, 1997	may, 1997
Abono intereses	3.806,00 Pts	2.532,00 Pts						
Abono ints. plazo						3.750,00 Pts		1.625,00 Pts
Carga mon. telef.						-5.000,00 Pts	-2.000,00 Pts	
Cargo comisiones				-50,00 Pts				
Cargo compra	-92.689,00 Pts	-130.943,00 Pts	-63.989,00 Pts	-53.476,00 Pts	-64.453,00 Pts	-43.095,00 Pts	-17.030,00 Pts	
Domiciliación	1.156.542,00 Pts							
Ingreso con libreta		202.400,00 Pts	295.000,00 Pts	190.000,00 Pts	210.000,00 Pts			690.000,00 Pts
Ingreso cheque c/ib			25.000,00 Pts					
Ingreso efectivo							240.000,00 Pts	
Orden transferencia	-75.700,00 Pts		-68.000,00 Pts	-68.000,00 Pts	-95.000,00 Pts	-68.000,00 Pts	-86.000,00 Pts	-86.000,00 Pts
Pago cajero autom.	-39.000,00 Pts	-104.000,00 Pts	-87.000,00 Pts	-43.000,00 Pts	-74.000,00 Pts	-47.000,00 Pts	-8.000,00 Pts	
Pago con libreta	-500.000,00 Pts	-75.000,00 Pts						
Pago rbo. luz		-5.067,00 Pts						
Pago rbo. telef.		-32.986,00 Pts			-43.571,00 Pts		-58.425,00 Pts	
Pago rbs. varios				-49.486,00 Pts	-18.363,00 Pts	-18.363,00 Pts	-18.363,00 Pts	-18.363,00 Pts
Sum	452.959,00 Pts	-143.064,00 Pts	51.475,00 Pts	-36.410,00 Pts	-43.066,00 Pts	3.117,00 Pts	562.232,00 Pts	

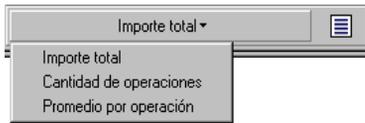
Como se observa en la imagen, en la dimensión horizontal aparece la fecha, mientras que la vertical corresponde a los distintos tipos de operaciones. Utilizando directamente la rejilla podemos intercambiar las dimensiones, mezclarlas, ocultarlas y restaurarlas. Mostramos a continuación, como ejemplo, el resultado de contraer la dimensión de la fecha; aparecen los gastos totales agrupados por conceptos:

Información sobre cuenta bancaria		Evolución del saldo						
		Rejilla de análisis						
		Gráfico de análisis						
Concepto	Fecha	nov, 1996	dic, 1996	ene, 1997	feb, 1997	mar, 1997	abr, 1997	may, 1997
Abono intereses	6.338,00 Pts	5.375,00 Pts	-7.000,00 Pts	-50,00 Pts	-465.675,00 Pts	1.156.542,00 Pts	1.587.400,00 Pts	25.000,00 Pts

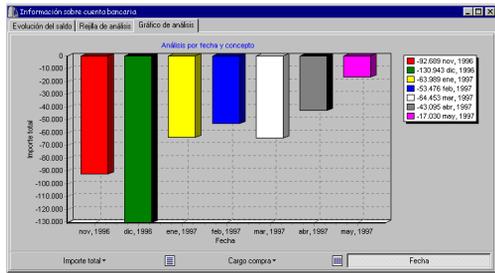
Sin embargo, es más sencillo utilizar el componente *TDecisionPivot* para manejar las dimensiones, y es lo que haremos en la siguiente página, añadiendo un *TDecisionGraph* y un pivote; el primero se alineará con *alClient* y el segundo con *alBottom*. Realmente, es necesario incluir pivotes para visualizar los gráficos de decisión, pues de otro modo no pueden modificarse dinámicamente las dimensiones de análisis. La siguiente figura muestra la tercera página de la aplicación en funcionamiento:



Mediante el pivote se puede cambiar el tipo de estadística que se muestra en el gráfico; para esto se utiliza el botón de la izquierda, que al ser pulsado muestra las funciones previstas durante el diseño de la consulta.



Otra operación importante consiste en ocultar dimensiones o en realizar la operación conocida como *drill in* (aproximadamente, una perforación). Hemos visto antes el resultado de ocultar la dimensión de la fecha: quedan entonces los totales, con independencia de esta dimensión. Ahora bien, al efectuar un *drill in* podemos mostrar exclusivamente los datos correspondientes a un valor determinado de la dimensión afectada. El *drill in* se activa pulsando el botón derecho del ratón sobre el botón de la dimensión que queremos modificar. La figura siguiente muestra exclusivamente el importe dedicado a *Cargo de compras*. Las barras están invertidas, pues los valores mostrados son negativos por corresponder a extracciones.



Por último, es conveniente saber que los gráficos de decisión cuentan con una amplia variedad de métodos de impresión. De esta forma, podemos dejar que el usuario de

nuestra aplicación elija la forma de visualizar los datos que le interesan y, seleccionando un comando de menú, pueda disponer en papel de la misma información que se le presenta en la pantalla del ordenador.

Por ejemplo, el siguiente método sirve para imprimir, de forma independiente, un gráfico de decisión, con la orientación apaisada:

```
void __fastcall TForm1::miImprimirClick(TObject *Sender)
{
    DecisionGraph1->PrintOrientation(poLandscape);
}
```

La constante *poLandscape* está definida en la unidad *Printers*, que debe ser incluida explícitamente en la unidad.

Uso y abuso de Decision Cube

El lector se habrá dado cuenta de que llevamos unas cuantas secciones sin escribir sino una sola línea de código. En realidad, la configuración de un cubo de decisión es una tarea muy sencilla. Pero esta misma sencillez lleva al programador a utilizar equivocadamente estos componentes. No voy a decirle cuáles son los usos “correctos” de Decision Cube, pues no me gustaría poner límites a su imaginación, pero sí voy a ilustrar algunos ejemplos realmente desacertados.

Comencemos por la interfaz visual. Hemos visto que *TDecisionGraph* admite la visualización de al menos dos dimensiones de forma simultánea. Si se quiere mostrar sólo una dimensión, obtendremos una serie (en el sentido que esta palabra tiene para *TChart* y *TDBChart*); si activamos dos dimensiones a la vez, se dibujará una sucesión de series. Y aquí tenemos el primer problema: a no ser que la segunda dimensión contenga un conjunto muy pequeño de valores posibles, el gráfico resultante será un galimatías, pues el usuario verá muchas series apiladas una sobre otras. Reto a cualquiera a que saque algo en claro de este tipo de gráficos.

Es posible configurar una dimensión para que contenga el valor *binSet* en su forma de agrupamiento, aunque no es posible cambiar este valor en tiempo de diseño. En la próxima sección mostraremos cómo hacerlo en tiempo de ejecución. Cuando este valor está activo, la dimensión siempre estará en el estado *drilled in*, y la cantidad de memoria asignada al cubo será mucho menor. Desgraciadamente, los componentes visuales, como *TDecisionPivot*, tienen problemas al enfrentarse a este tipo de dimensiones, y la misma funcionalidad puede obtenerse mediante consultas paramétricas.

Más problemas con la cardinalidad de las dimensiones: he visto un programa que intentaba usar como dimensión nada menos que un nombre de cliente²⁴. ¿Qué tiene de malo? En primer lugar, que se generan demasiadas celdas para que el Cubo pueda manejarlas. Aunque Borland no ha hecho público el código fuente de Decision Cube con los detalles de implementación, sabemos que este componente utiliza una estructura de datos en memoria cuyo tamaño es proporcional a la multiplicación de el número de valores por cada dimensión. Lo más probable es que el Decision Cube “corte” la dimensión en algún sitio y solamente muestre un pequeño número de clientes. Precisamente esto era lo que pasaba con el ejemplo que mencioné al comenzar el párrafo.

En segundo lugar, ¿qué información útil, qué tendencia podemos descubrir en un cubo que una de sus dimensiones sea el nombre o código de cliente? Supongamos incluso que utilizamos esa única dimensión. ¿Cuál sería el resultado que mostraría un *TDecisionGrid*? Uno similar al de un *TDBGrid*, y está claro que la rejilla de datos de toda la vida lo haría mejor. ¿Y sobre un *TDecisionGraph*? Tendríamos una serie que, a simple vista, semejaría una función discontinua y caótica; la información gráfica no tendría valor alguno. Y en cualquier caso, un sencillito *TDBChart* sería mucho más útil y eficiente.

Modificando el mapa de dimensiones

Las posibilidades de configuración dinámica de un cubo de decisión van incluso más allá de las que ofrece directamente el componente de pivote. Por ejemplo, puede interesarnos el cambiar el agrupamiento de una dimensión de tipo fecha. En vez de agrupar los valores por meses, queremos que el usuario decida si las tendencias que analiza se revelan mejor agrupando por trimestres o por años.

El siguiente procedimiento muestra como cambiar el tipo de agrupamiento para determinado elemento de un cubo de decisión, dada la posición del elemento:

```
void __fastcall ModificarCubo(TDecisionCube *ACube, int ItemNo,
    TBinType Grouping)
{
    TCubeDims *DM = new TCubeDims(0, __classid(TCubeDim));
    try
    {
        DM->Assign(ACube->DimensionMap);
        DM->Items[ItemNo]->BinType = Grouping;
        ACube->Refresh(DM, False);
    }
}
```

²⁴ En esta discusión asumiré que se trata de una tabla de clientes grande, que es lo típico.

```

    _finally
    {
        delete DM;
    }
}

```

El parámetro *ItemNo* indica la posición de la dimensión dentro del cubo, mientras que *Grouping* es el tipo de agrupamiento que vamos a activar: *binMonth*, *binQuarter* ó *binYear*. Estas constantes están definidas en la unidad *MxCommon*. También puede utilizar el valor *binSet*, para que la propiedad aparezca en estado *drilled in*, siempre que también asignemos algún valor a la propiedad *StartValue* o *StartDate* de la dimensión.

El programador puede incluir algún control para que el usuario cambie el agrupamiento, por ejemplo, un combo, e interceptar el evento *OnChange* del mismo para aprovechar y ejecutar el procedimiento antes mostrado.

Esta técnica a veces provoca una excepción en C++ Builder 4. El fallo puede reproducirse en tiempo de diseño fácilmente: basta con crear un formulario basado en Decision Cube e intentar cerrar el conjunto de datos de origen. Después de mucho experimentar con estos controles, he notado que la probabilidad de fallo disminuye si no existe un *TDecisionGraph* a la vista en el momento del cambio de dimensiones.

Descenso a los abismos

ATODOS LOS DESARROLLADORES NOS ATRAE LA idea de especializarnos en algún área esotérica y confusa de la programación, para poder después presumir de “expertos en algo”. Nadie está completamente libre de esta superstición. Al programador de aplicaciones para bases de datos con C++ Builder puede parecerle interesante descender en un escalón de abstracción y acceder a las profundas y oscuras entrañas del BDE, buscando más control y eficiencia. ¿Hasta qué punto le será útil este esfuerzo?

Hace un par de años, le hubiera recomendado ciegamente que aprendiera al menos los detalles básicos del manejo del API del BDE. Pero a estas alturas no lo tengo tan claro: las sucesivas versiones de la VCL han terminado por encapsular la mayoría de las tareas interesantes del Motor de Datos. Las principales áreas que quedan por cubrir en la VCL son:

- Algunas funciones de recuperación y modificación de parámetros de configuración.
- Creación y reestructuración de tablas de Paradox y dBase.
- Algunas posibilidades de ciertas funciones de respuesta.

Estas técnicas nos servirán de excusa para explicar cómo se programa directamente con el Motor de Datos de Borland.

Inicialización y finalización del BDE

Lo primero es saber cómo tener acceso a las funciones del API del BDE. Todas las declaraciones del Motor de Datos se encuentran en el fichero *bde.hpp*, del directorio *include\vcl* de C++ Builder. Como el directorio *vcl* se encuentra en la lista de directorios donde C++ Builder busca automáticamente cabeceras, basta con la siguiente directiva para tener acceso a las declaraciones del BDE:

```
#include <bde.hpp>
```

A continuación, hay que aprender a iniciar y descargar de memoria las DLLs del BDE. Cuando trabajamos con la VCL, estas tareas son realizadas automáticamente por la clase *TSession*. Ahora debemos utilizar explícitamente la siguiente función:

```

struct DBIEnv {
    char szWorkDir[261];
    char szIniFile[261];
    Word bForceLocalInit;
    char szLang[32];
    char szClientName[32];
};
typedef DBIEnv *pDBIEnv;

Word __stdcall DbiInit(pDBIEnv pEnv);

```

Estas declaraciones han sido extraídas de *bde.hpp*. La ayuda en línea puede dar una versión diferente de algunos tipos de datos. Por ejemplo, según la ayuda, la declaración de *DbiInit* debe ser:

```

DBIResult DBIFN DbiInit(pDBIEnv pEnv);

```

Pero es fácil hacer corresponder el tipo de retorno *DBIResult* con el tipo *Word*, y la macro *DBIFN* con la directiva **__stdcall**. En cualquier caso, lo importante es que todas las funciones del BDE devuelven un valor entero para indicar si pudieron cumplir o no con su cometido. En la siguiente sección veremos más detalles acerca del control de error.

La forma más sencilla de llamar a *DbiInit* es pasando el puntero nulo como parámetro, en cuyo caso se inicializa el BDE con valores por omisión:

```

// Inicializar el BDE con parámetros por omisión
DbiInit(NULL);

```

Si utilizamos un puntero a una variable de entorno (*DBIEnv*) podemos indicar:

Campo	Significado
<i>szWorkDir</i>	El directorio de trabajo.
<i>szIniFile</i>	El fichero de configuración.
<i>bForceLocalInit</i>	Obliga a una inicialización local.
<i>szLang</i>	El idioma a utilizar.
<i>szClientName</i>	Una identificación para la aplicación.

Quizás el más importante de estos parámetros sea *szLang*, que indica el idioma del usuario, pero ya he explicado su uso en el capítulo 30.

Son tres los posibles valores de retorno de *DbiInit*:

Valor de retorno	Significado
<i>DBIERR_NONE</i>	Hakuna matata
<i>DBIERR_MULTIPLEINIT</i>	El BDE ya había sido inicializado
<i>DBIERR_OSACCESS</i>	No hay permisos de escritura sobre el directorio actual

Si la función devuelve *DBIERR_MULTIPLEINIT*, no tiene por qué preocuparse, pues de todos modos puede utilizar las funciones del BDE. Esto es precisamente lo que sucedería si inicializáramos manualmente el BDE en una aplicación de C++ Builder que utilizase los componentes de acceso a datos de la VCL: la inicialización de la VCL fallaría con el error mencionado, pero no se consideraría un error serio.

Para finalizar el trabajo con el BDE debemos llamar a la siguiente función:

```
Word __stdcall DbExit(void);
```

Según la documentación del BDE, si lo inicializamos desde una DLL estamos obligados a llamar la función *DbiDllExit* antes de llamar a *DbExit*.

El control de errores

Como hemos visto, las funciones del BDE utilizan códigos de error para señalar fallos de contrato. Un lenguaje moderno, sin embargo, debe utilizar funciones para este propósito. Por lo tanto, la VCL utiliza la función *Check*, definida en la unidad *DBTables*, para verificar el código de retorno de las funciones del Motor de Datos, y lanzar una excepción si detecta que algo va mal:

```
Check(DbInit(NULL));
```

Check lanza una excepción de clase *EDBEngineError*. Ya hemos explicado, en el capítulo 28, la estructura de esta clase de excepción, y vimos que informa acerca de una pila de errores: el error producido por el servidor SQL (de ser aplicable), la interpretación del BDE, etc. Siempre que pueda, utilice *Check* para el control de errores.

No obstante, puede ser que escribamos en algún momento una pequeña aplicación y no deseemos cargar con nada declarado por la VCL para mantener una huella pequeña en memoria. ¿Cómo extrae *Check* la información compleja que le sirve para construir el objeto *EDBEngineError*? Estas son las funciones del BDE para el manejo de errores:

```
Word __stdcall DbGetErrorString(Word rslt, char *pszError);
Word __stdcall DbGetErrorContext(short eContext, char *pszContext);
Word __stdcall DbGetErrorInfo(BOOL bFull, DBIErrInfo &ErrInfo);
Word __stdcall DbGetErrorEntry(Word uEntry, int &ulNativeError,
char *pszError);
```

La más sencilla es *DbiGetErrorString*, que traduce un código de error a su representación textual. Si necesita un sustituto barato de *Check*, puede utilizar la siguiente función:

```
void __fastcall CheapCheck(Word rslt)
{
    if (rslt != DBIERR_NONE)
    {
        char buffer[DBIMAXMSGLEN + 1];
        DbiGetErrorString(rslt, buffer);
        throw Exception(buffer);
    }
}
```

Las restantes funciones deben llamarse inmediatamente después de producirse el error, porque la información que extraen siempre se refiere al último error. Es muy interesante, por ejemplo, la función *DbiGetErrorContext*, pues nos permite averiguar determinadas circunstancias acerca de ese último error. El parámetro *eContext* indica si queremos conocer qué tabla, qué índice, qué usuario ... está involucrado en el fallo:

```
char buffer[DBIMAXMSGLEN + 1];
DbiGetErrorContext(ecINDEXNAME, buffer);
ShowMessage("Indice: " + AnsiString(buffer));
```

Sin embargo, no podemos aprovechar las habilidades de esta función en un manejador del evento *OnPostError*, pues al llegar a ese punto la VCL ya ha eliminado la información de contexto del error generado.

Sesiones y conexiones a bases de datos

Repasemos mentalmente la lista de componentes de la VCL para acceso a datos, ¿recuerdas las sesiones? ¿Cómo se administran las sesiones en este nivel de programación? Para aliviar la vida del programador, el BDE reconoce que no es frecuente trabajar con varias sesiones simultáneas, e implementa una sesión activa. Existe una sesión por omisión, que el BDE crea y administra por su cuenta, pero podemos crear sesiones adicionales y activarlas cuando lo consideremos necesario.

```
Word __stdcall DbiStartSession(char* pszName, hDBISes &hSes,
    char *pNetDir);
Word __stdcall DbiCloseSession(hDBISes hSes);
Word __stdcall DbiGetCurrSession(hDBISes &hSes);
Word __stdcall DbiSetCurrSession(hDBISes hSes);
```

Como no es necesario utilizar sesiones explícitamente para las tareas habituales con el BDE, no insistiré más en el asunto.

Si seguimos adelante con la jerarquía de objetos del BDE, tropezamos de narices con las conexiones a bases de datos: el equivalente aproximado de la *TDatabase* de nuestra VCL. La función que abre una base de datos es la siguiente:

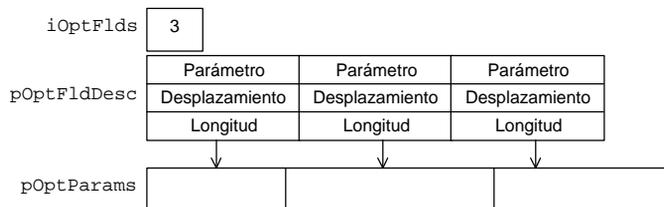
```
Word __stdcall DbiOpenDatabase(char *pszDbName, char *pszDbType,
    DBIOpenMode eOpenMode, DBIShareMode eShareMode,
    char *pszPassword, Word iOptFlds, pFLDDesc pOptFldDesc,
    void *pOptParams, hDBIDb &hDb);
```

Es típico del BDE utilizar un montón de parámetros incluso en las funciones más elementales, aunque para la mayoría de los casos baste el uso de valores por omisión. Por tal motivo, es preferible demostrar con ejemplos el uso de este tipo de funciones. Para abrir una base de datos de Paradox o dBase, y asociarla a un determinado directorio, utilice una función como la siguiente:

```
hDBIDb __fastcall AbrirDirectorio(AnsiString Directorio)
{
    hDBIDb rslt;
    CheapCheck(DbiOpenDatabase(0, 0, dbiREADWRITE, dbiOPENSERIALIZED,
        0, 0, 0, 0, rslt));
    CheapCheck(DbiSetDirectory(rslt, Directorio.c_str()));
    return rslt;
}
```

La base de datos, claro está, se crea en el contexto definido por la sesión activa. Recuerde que *CheapCheck* ha sido definido por nosotros, para evitar las referencias a la VCL.

Si hay que crear una conexión no persistente a una base de datos SQL con parámetros explícitos, se complica el uso de *DbiOpenDatabase*. El siguiente esquema intenta aclarar la forma de suministrar dichos valores, a través de los parámetros *iOptFlds*, *pOptFldDesc* y *pOptParams*:



El tipo *FLDDesc* sirve en realidad para describir un campo, y volveremos a encontrarlo al estudiar la creación de tablas. En esta función, sirve para describir un parámetro, y solamente nos interesan tres de sus campos: *szName*, para el nombre del parámetro de conexión, *iOffset*, para especificar el desplazamiento dentro del *buffer* apuntado por *pOptParams* donde se encuentra el valor, e *iLen*, para indicar la longitud

El primer parámetro contiene el *handle* a una conexión de base de datos, el segundo especifica qué debe suceder si ya existe una tabla con ese nombre, mientras que en el tercero se pasa por referencia una gigantesca estructura que contiene todos los detalles necesarios para crear la tabla. La declaración de *CRTblDesc* es como sigue:

```

struct CRTblDesc {
    char szTblName[261], szTblType[32], szErrTblName[261];
    char szUserName[32], szPassword[32];
    Word bProtected;           // Sólo si es Paradox
    Word bPack;
    Word iFldCount;           // Descriptores de campos
    CROpType *pecrFldOp;
    FLDDesc *pfldDesc;
    Word iIdxCount;           // Descriptores de índices
    CROpType *pecrIdxOp;
    IDXDesc *pidxDesc;
    Word iSecRecCount;        // Descriptores de seguridad
    CROpType *pecrSecOp;
    SECDesc *psecDesc;
    Word iValChkCount;        // Restricciones y validaciones
    CROpType *pecrValChkOp;
    VCHKDesc *pvchkDesc;
    Word iRintCount;          // Integridad referencial
    CROpType *pecrRintOp;
    RINTDesc *printDesc;
    Word iOptParams;          // Parámetros opcionales
    FLDDesc *pfldOptParams;
    void *pOptData;
};

```

No se asuste, amigo, que no es necesario rellenar todos los campos de la estructura anterior. Para empezar, todos los atributos de tipo puntero a *CROpType* se emplean solamente durante la reestructuración de una tabla. En segundo lugar, observe que hay muchos pares de parámetros dependientes entre sí, como *iFldCount*, que contiene el número de campos a crear, y *pfldDesc*, que apunta a un vector con las descripciones de estos campos.

Ya nos hemos tropezado con el tipo *FLDDesc*, que sirve para la descripción de campos. Ahora presentaré su definición completa:

```

struct FLDDesc {
    Word iFldNum;             // Número secuencial del campo
    char szName[32];          // Nombre del campo
    Word iFldType;            // Tipo del campo
    Word iSubType;            // Subtipo
    short iUnits1, iUnits2;   // Tamaño, o precisión/escala
    Word iOffset, iLen, iNullOffset;
    FLDVchk efldvVchk;        // ¿Tiene validaciones?
    FLDRights efldrRights;    // Derechos sobre el campo
    Word bCalcField;          // ¿Es calculado?
    Word iUnUsed[2];
};

```

Hay que aclarar que la mayoría de los atributos, a partir de *iOffset*, no se utilizan con *DbiCreateTable*, sino que sirven para recuperar información sobre campos de tablas existentes.

El otro tipo importante es *IDXDdesc*, el descriptor de índices, cuya declaración mostramos a continuación:

```

struct IDXDdesc {
    char szName[261];           // Nombre del índice
    Word iIndexId;             // Número secuencial
    char szTagName[32];       // Nombre del tag (si es MDX/CDX)
    char szFormat[32];
    Word bPrimary, bUnique;
    Word bDescending, bMaintained, bSubset;
    Word bExpIdx, iCost, iFldsInKey, iKeyLen, bOutOfDate;
    Word iKeyExpType;
    Word aiKeyFld[16];
    char szKeyExp[221];
    char szKeyCond[221];
    Word bCaseInsensitive;
    Word iBlockSize;
    Word iRestrNum;
    Word abDescending[16];
    Word iUnUsed[16];
};

```

También hay un tipo para expresar restricciones de integridad a nivel de campo: campos requeridos, mínimos, máximos y valores por omisión.

```

struct VCHKDesc {
    Word iFldNum;              // Número del campo al que se aplica
    Word bRequired, bHasMinVal, bHasMaxVal, bHasDefVal;
    Byte aMinVal[256];        // Representación del valor mínimo
    Byte aMaxVal[256];        // Representación del valor máximo
    Byte aDefVal[256];        // Valor por omisión
    char szPict[176];         // Máscara (sólo Paradox)
    LKUPType elkupType;       // Campos lookup (sólo Paradox)
    char szLkupTblName[261];
};

```

Curiosamente, con las últimas versiones del BDE no se puede crear una restricción de integridad referencial a la vez que se crea la tabla, sino que hay que añadir estas restricciones con una llamada a *DbiDoRestructure*, la función de modificación de esquemas que veremos en la siguiente sección. De este modo, se simplifica aún más la creación de tablas.

En vez de largar una docta parrafada acerca de cada uno de los atributos de las estructuras anteriores, es preferible un pequeño ejemplo que muestre el uso típico de las mismas. Cuando queremos crear una tabla y conocemos perfectamente la estructura de la misma en tiempo de diseño, es frecuente que los descriptores de campos,

índices y de validaciones se almacenen en vectores estáticos, como se muestra en el siguiente fragmento de código:

```
// Descriptores de campos
FLDDesc fields[] = {
    {1, "Codigo", fldDBLONG, fldUNKNOWN, 0, 0},
    {2, "Nombre", fldDBCHAR, fldUNKNOWN, 30, 0},
    {3, "Alta", fldDBDATETIME, fldUNKNOWN, 0, 0}};

// Validaciones
VCHKDesc checks[] = {
    {2, TRUE, FALSE, FALSE, FALSE},
    {3, FALSE, FALSE, FALSE, TODAYVAL}};

// Descriptores de índices
IDXDesc indexes[] = {
    // Primario, Unico
    {"", 1, "Primario", "", 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, {0},
    "CODIGO"},
    // Secundario, Unico
    {"", 2, "ByName", "", 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, {0},
    "NOMBRE"}};

void __fastcall TForm1::Button1Click(TObject *Sender)
{
    CRTblDesc TableDesc;

    memset((void*)&TableDesc, 0, sizeof(TableDesc));
    strcpy(TableDesc.szTblName, "Clientes.DBF");
    strcpy(TableDesc.szTblType, szDBASE);
    TableDesc.iFldCount = 3;
    TableDesc.pfldDesc = fields;
    TableDesc.iIdxCount = 2;
    TableDesc.pidxDesc = indexes;
    TableDesc.iValChkCount = 2;
    TableDesc.pvchkDesc = checks;
    Check(DbCreateTable(Database1->Handle, 1, TableDesc));
}
```

El método anterior crea una tabla en formato dBase, con tres campos, de tipos entero, carácter y fecha/hora, respectivamente. El segundo campo es no nulo, mientras que el tercero utiliza la fecha actual como valor por omisión. Observe que el primer atributo de cada *VCHKDesc*, de nombre *iFldNum*, se refiere al número de secuencia de determinado campo, que se indica en el atributo homónimo de la estructura *FLDDesc*. También se crea una clave primaria sobre la primera columna, y una clave secundaria sobre la segunda.

Reestructuración

La otra función que utiliza la estructura *CRTTblDesc* es *DbiDoRestructure*, y sirve para modificar la definición de una tabla:

```
Word __stdcall DbiDoRestructure(hDBIDb hDb, Word iTblDescCount,
    pCRTblDesc pTblDesc, char *pszSaveAs, char *pszKeyviolName,
    char *pszProblemsName, BOOL bAnalyzeOnly);
```

En teoría, esta función puede reestructurar simultáneamente varias tablas en una misma base de datos, cuyo *handle* se pasa en *hDb*. En la práctica, el segundo parámetro debe ser siempre *1*, para indicar que sólo se va a modificar una tabla. En el parámetro *pTblDesc* se pasa la descripción de las operaciones a efectuar sobre la pobre tabla. Podemos también cambiar el nombre a la tabla, guardar información sobre las violaciones de la clave primaria en otra tabla y quedarnos con una copia de los registros con problemas. El último parámetro, *bAnalyzeOnly*, no funciona y tiene toda la pinta de que nunca llegará a funcionar.

Para mostrar el uso de *DbiDoRestructure* implementaremos una de las restricciones que no pueden crearse mediante métodos de alto nivel con C++ Builder: las restricciones de integridad referencial de dBase. Pero primero necesitamos una tabla de detalles. En la sección anterior habíamos creado una tabla de clientes genérica; ahora supondremos que estos clientes realizan consultas técnicas, y que necesitamos una tabla *Consultas* para almacenarlas. Esta segunda tabla debe tener un campo *Cliente*, en el cual guardaremos el código del cliente que hace la consulta:

```
// Creación de una tabla de detalles

FLDDesc fields1[] = {
    {1, "Codigo", fldDBAUTOINC, fldUNKNOWN, 0, 0},
    {2, "Cliente", fldDBLONG, fldUNKNOWN, 0, 0},
    {3, "Fecha", fldDBDATETIME, fldUNKNOWN, 0, 0},
    {4, "Asunto", fldDBCHAR, fldUNKNOWN, 30, 0},
    {5, "Texto", fldDBMEMO, fldUNKNOWN, 0, 0}};

VCHKDesc checks1[] = {
    {2, TRUE, FALSE, FALSE, FALSE},
    {3, TRUE, FALSE, FALSE, FALSE},
    {4, FALSE, FALSE, FALSE, TODAYVAL}};

IDXDesc indexes1[] = {
    // Primario, Unico
    {"", 1, "Primario", "", 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, {0},
     "CODIGO"},
    // Secundario
    {"", 2, "Cliente", "", 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, {0},
     "CLIENTE"}};

void __fastcall TForm1::Button2Click(TObject *Sender)
{
    CRTblDesc TableDesc;

    memset((void*)&TableDesc, 0, sizeof(TableDesc));
    strcpy(TableDesc.szTblName, "Consultas.DBF");
    strcpy(TableDesc.szTblType, szDBASE);
    TableDesc.iFldCount = 5;
    TableDesc.pfldDesc = fields1;
    TableDesc.iIdxCount = 2;
    TableDesc.pidxDesc = indexes1;
```

```

TableDesc.iValChkCount = 3;
TableDesc.pvchkDesc = checks1;
Check(DbiCreateTable(Database1->Handle, 1, TableDesc));
}

```

Una vez que está creada la tabla de detalles, podemos añadirle entonces la restricción de integridad:

```

// Añadir la restricción de integridad referencial
CROpType Operations[] = {crADD};

RINTDesc rInts[] = {{1, "RefCliente", rintDEPENDENT, "Clientes.DBF",
    rintRESTRICT, rintRESTRICT, 1, {2}, {1}}};

void __fastcall TForm1::Button3Click(TObject *Sender)
{
    CRTblDesc TableDesc;

    memset((void*)&TableDesc, 0, sizeof(TableDesc));
    strcpy(TableDesc.szTblName, "Consultas.DBF");
    strcpy(TableDesc.szTblType, szDBASE);
    TableDesc.pcrRintOp = Operations;
    TableDesc.iRintCount = 1;
    TableDesc.printDesc = rInts;
    Check(DbiDoRestructure(Database1->Handle, 1, &TableDesc,
        0, 0, 0, FALSE));
}

```

Observe que debemos rellenar el atributo *pcrRintOp* del descriptor de tablas. En general podemos añadir, eliminar o modificar simultáneamente varias restricciones, aunque aquí solamente añadimos una sola.

Eliminación física de registros borrados

Una aplicación muy especial de *DbiDoRestructure* es la eliminación física de registros borrados en Paradox y dBase, y la reconstrucción de sus índices. En tal caso, basta con indicar el nombre de la tabla y asignar *True* al atributo *bPack* del descriptor de operaciones:

```

void __fastcall PackParadox(const AnsiString ADatabase,
    const AnsiString ATable)
{
    CRTblDesc PdxStruct;
    TDatabase *ADB = Session->OpenDatabase(ADatabase);
    try {
        memset((void*) &PdxStruct, 0, sizeof(PdxStruct));
        strcpy(PdxStruct.szTblName, ATable.c_str());
        PdxStruct.bPack = True;
        Check(DbiDoRestructure(ADB->Handle, 1, &PdxStruct,
            0, 0, 0, False));
    }
}

```

```

    __finally {
        Session->CloseDatabase(ADB);
    }
}

```

Pero existe una función, *DbiPackTable*, que puede utilizarse específicamente para dBase. Esta vez, la función necesita que el cursor de la tabla esté abierto:

```

void __fastcall PackDBF(const AnsiString ADatabase,
    const AnsiString ATable)
{
    std::auto_ptr<TTable> t(new TTable(0));
    t->DatabaseName = ADatabase;
    t->TableName = ATable;
    t->Exclusive = True;
    t->Open();
    Check(DbiPackTable(t->Database->Handle, t->Handle, 0, 0, True));
}

```

También existe la función *DbiRegenIndexes*, que reconstruye los índices de tablas de Paradox y dBase, sin afectar a los registros eliminados lógicamente. Al igual que sucede con *DbiPackTable*, es necesario que la tabla esté abierta en modo exclusivo:

```

void __fastcall Reindexar(const AnsiString ADatabase,
    const AnsiString ATable)
{
    std::auto_ptr<TTable> t(new TTable(0));
    t->DatabaseName = ADatabase;
    t->TableName = ATable;
    t->Exclusive = True;
    t->Open();
    Check(DbiRegenIndexes(t->Handle));
}

```

Cursores

Una vez que tenemos acceso a una base de datos, contamos con todo lo necesario para abrir una tabla o una consulta para recorrer sus filas. Para representar un conjunto de datos abierto, o *cursor*, se utiliza el tipo de datos *hDBICur*, que contiene el *handle* del cursor. Existen muchos tipos de cursores, y por lo tanto, muchas funciones del BDE que abren un cursor. Las más “tradicionales”, que sirven para abrir tablas y consultas, son las siguientes:

Función	Propósito
<i>DbiOpenTable</i>	Abre un cursor para una tabla
<i>DbiOpenInMemTable</i>	Crea y abre una tabla en memoria
<i>DbiQExecDirect</i>	Ejecuta una consulta sin parámetros
<i>DbiQExec</i>	Ejecuta una consulta previamente preparada.

Pero también existen funciones que abren tablas “virtuales”, que realmente no están almacenadas como tales, pero que sirven para devolver con el formato de un conjunto de datos información de longitud variable. Si quiere conocer cuáles son todas estas funciones, vaya al índice de la ayuda en línea del API del BDE, y teclee *DbiOpenList functions*, como tema de búsqueda.

¿Tiene sentido entrar a fondo en el estudio de las funciones de apertura y navegación sobre cursores? ¡No, absolutamente! La VCL encapsula bastante bien esta funcionalidad mediante las clases *TBDEDataSet* y *TDBDataSet*, como para tener que repetir esta labor. Se trata, además, de funciones complejas, como puede deducirse de la declaración de *DbiOpenTable* (¡12 parámetros!):

```
Word __stdcall DbiOpenTable(hDBIDb hDb, char* pszTableName,
    char* pszDriverType, char* pszIndexName, char* pszIndexTagName,
    Word iIndexId, DBIOpenMode eOpenMode, DBIShareMode eShareMode,
    XLTMode exltMode, BOOL bUniDirectional, void* pOptParams,
    hDBICur &hCursor);
```

En cualquier caso, si necesita utilizar alguno de los cursores especiales, es preferible derivar un nuevo componente a partir de *TBDEDataSet*, si no necesita una conexión a una base de datos, o de *TDBDataSet*, en caso contrario. Supongamos que queremos obtener en formato tabular las restricciones de integridad referencial asociadas a una tabla. El BDE nos ofrece la función *DbiOpenRintList* para abrir un cursor con dicha información:

```
Word __stdcall DbiOpenRintList(hDBIDb hDb, char* pszTableName,
    char* pszDriverType, hDBICur& hChkCur);
```

Pero nosotros, gente inteligente, creamos la siguiente clase:

```
class TRITable : public TDBDataSet
{
private:
    AnsiString FTableName;
protected:
    hDBICur __fastcall CreateHandle();
public:
    __fastcall TRITable(TComponent *aOwner) : TDBDataSet(aOwner) {}
__published:
    __property AnsiString TableName =
        {read = FTableName, write = FTableName};
};
```

La única función redefinida tiene una implementación trivial:

```
hDBICur __fastcall TRITable::CreateHandle()
{
    hDBICur rslt;
    Check(DbiOpenRintList(DBHandle, FTableName.c_str(), 0, rslt));
    return rslt;
}
```

La siguiente imagen demuestra el funcionamiento del componente anterior dentro de una sencilla aplicación:

REFINTNUM	NAME	TYPE	OTHERTABLE	MODIFYQUAL	DELETEQUAL	FIELDCOUNT
	DEMO.SYS_C00589	1	DEMO.SALES_ORDER			1
	DEMO.SYS_C00590	1	DEMO.PRODUCT			1

El objetivo de incluir la imagen es mostrar cómo, gracias a la maquinaria interna del tipo *TDBDataSet*, no tenemos que preocuparnos por la definición de los campos; estas definiciones se extraen automáticamente de las propiedades del cursor abierto.

Un ejemplo de iteración

De todos modos, es provechoso mostrar al menos un ejemplo básico de iteración sobre un cursor. Ahora echaremos mano de la siguiente función:

```
Word __stdcall DbiOpenCfgInfoList(hDBICfg hCfg,
    DBIOpenMode eOpenMode, CFGMode eConfigMode, char* pszCfgPath,
    hDBICur &hCur);
```

El primer parámetro siempre debe ser el puntero nulo, mientras que el tercero también debe llevar el valor obligado *cfgPersistent*. En el segundo parámetro indicamos si queremos abrir este cursor para lectura o también para escritura. En el cuarto se pasa una cadena con un formato de directorio. El BDE almacena su información de configuración en forma jerárquica, y este parámetro se refiere a un nodo determinado de la jerarquía. Cuando abrimos un cursor para un nodo, obtenemos una colección de registros con el formato (*parámetro, valor*).

Con esta información a nuestro alcance, es fácil diseñar una función que, dada una de estas rutas y el nombre de un parámetro, devuelva el valor asociado:

```
AnsiString __fastcall GetBDEInfo(const AnsiString Path,
    const AnsiString Param)
{
    hDBICur hCur;
    CFGDesc Desc;

    DbiOpenCfgInfoList(NULL, dbiReadOnly, cfgPersistent,
        Path.c_str(), &hCur);
    try {
        while (DbiGetNextRecord(hCur, dbiNoLock, &Desc, 0) ==
            DBIERR_NONE)
            if (strcmp(Desc.szNodeName, Param.c_str()) == 0)
                return AnsiString(Desc.szValue);
    }
}
```

```

    __finally {
        DbcCloseCursor(&hCur);
    }
    return "<Not found>";
}

```

El núcleo de la función es la iteración sobre el cursor mediante la función:

```

Word __stdcall DbiGetNextRecord(hDBICur hCursor, DBILockType eLock,
    void* pRecBuff, pRECPProps precProps);

```

En su tercer parámetro debemos pasar un puntero al *buffer* que recibe los datos. El formato de este *buffer* depende del tipo de cursor abierto, en particular, de las columnas que posea, pero para cada cursor especial del BDE ya existe un registro que puede recibir las filas del cursor. En el caso de *DbiOpenCfgInfoList*, el tipo apropiado es la estructura *CFGDesc*.

De todos modos, hay también un método de *TSession* que hace más o menos lo mismo:

```

void __fastcall TSession::GetConfigParams(const AnsiString Path,
    const AnsiString Section, TStrings* List);

```

Pero con la función del BDE tenemos la posibilidad de modificar cualquiera de sus parámetros, si utilizamos esta variante del algoritmo:

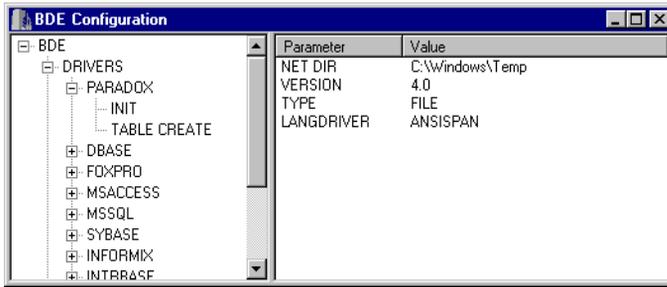
```

void __fastcall SetBDEInfo(const AnsiString Path,
    const AnsiString Param, const AnsiString Value)
{
    hDBICur hCur;
    CFGDesc Desc;

    DbiOpenCfgInfoList(0, dbiReadWrite, cfgPersistent,
        Path.c_str(), &hCur);
    try {
        while (DbiGetNextRecord(hCur, dbiNoLock, &Desc, 0) ==
            DBIERR_NONE)
            if (stricmp(Desc.szNodeName, Param.c_str()) == 0)
            {
                strcpy(Desc.szValue, Value.c_str());
                DbiModifyRecord(hCur, &Desc, True);
                break;
            }
    }
    __finally {
        DbcCloseCursor(&hCur);
    }
}

```

En el CD-ROM que acompaña al libro hemos incluido una pequeña aplicación que sirve para explorar la estructura arbórea de los parámetros de configuración del BDE. También se pueden cambiar los valores de los parámetros:



Las siguientes funciones especiales aprovechan los procedimientos anteriores para lograr de una forma más sencilla la modificación de dos de los parámetros del BDE más populares:

```
void __fastcall SetLocalShare(const AnsiString Value)
{
    SetBDEInfo("\\SYSTEM\\INIT", "LOCAL SHARE", Value);
}

void __fastcall SetNetDir(const AnsiString Value)
{
    SetBDEInfo("\\DRIVERS\\PARADOX\\INIT", "NET DIR", Value);
}
```

Propiedades

En el ejemplo anterior hemos leído los datos del registro activo dentro de un *buffer* de longitud fija, cuya estructura es suministrada por el BDE. ¿Y cómo hacemos en el caso general para averiguar la longitud necesaria del *buffer* de un cursor arbitrario? La respuesta es: leyendo las propiedades del cursor.

En realidad, casi todos los objetos del BDE tienen propiedades que pueden leerse y, en ocasiones, ser modificadas. Las propiedades del BDE son un mecanismo primitivo de asociar atributos a objetos cuya verdadera estructura interna está fuera del alcance del programador. Las funciones que manipulan propiedades son:

```
Word __stdcall DbiGetProp(hDBIObj hObj, int iProp, void* PropValue,
    Word iMaxLen, Word &iLen);
Word __stdcall DbiSetProp(hDBIObj hObj, int iProp, int iPropValue);
```

El objeto se indica en el primer parámetro, pero el más importante de los parámetros es *iProp*, que identifica qué propiedad queremos leer o escribir. Cada tipo diferente de objeto tiene su propio conjunto de propiedades válidas. Busque *objects*, *properties* en la ayuda en línea del BDE para un listado exhaustivo de todas las posibles propiedades. El siguiente ejemplo muestra cómo recuperar el nombre de una tabla asociada a un cursor:

```

Word iLen;
DBITBLNAME tblName;

DbiGetProp(((TBDEDataSet*)DataSet)->Handle, curTABLENAME,
           (void*) tblName, sizeof(tblName), iLen);
ShowMessage(AnsiString(tblName));

```

Cuando el objeto es precisamente un cursor, las propiedades más interesantes pueden obtenerse agrupadas mediante la función *DbiGetCursorProps*:

```

Word __stdcall DbiGetCursorProps(hDBICur hCursor,
                                CURProps &curProps);

```

La estructura *CURProps* tiene 36 atributos, incluyendo el inevitable *iUnUsed*, así que no le voy a aburrir contándole para qué sirve cada uno. Solamente le diré que el tamaño del *buffer* para el registro se puede obtener por medio del atributo *iRecBufSize*. En definitiva, esto era lo que queríamos averiguar al principio de esta sección, ¿no?

Para terminar, quiero mostrarle alguna técnica con propiedades que *realmente* merezca la pena. Uno de los principales trucos con los cursores del BDE tiene que ver con la posibilidad de limitar el número máximo de filas que éste puede retornar al cliente. Cuando estudiamos la configuración del Motor de Datos vimos el parámetro *MAX ROWS*, común a la mayoría de los controladores SQL. Si modificamos su valor, afectaremos a todas las tablas y consultas que se abran por medio del alias modificado. Sin embargo, podemos hacer que la limitación solamente se aplique a un cursor determinado.

```

void __fastcall LimitRows(TDBDataSet* DataSet, long MaxRows)
{
    char DBType[DBIMAXNAMELEN];
    Word Len;

    Check(DbiGetProp(hDBIObj(DataSet->DBHandle), dbDATABASETYPE,
                   DBType, sizeof(DBType), Len));
    if (strcmp(DBType, "STANDARD") == 0)
        throw EDBEngineError(DBIERR_NOTSUPPORTED);
    Check(DbiValidateProp(hDBIObj(DataSet->Handle), curMAXROWS,
                        True));
    Check(DbiSetProp(hDBIObj(DataSet->Handle), curMAXROWS, MaxRows));
}

```

Aunque la propiedad *curMAXROWS* puede aplicarse también a una tabla, la limitación se aplica a cada una de las consultas que ésta abre tras el telón, por lo cual el comportamiento de la tabla parecerá errático al usuario. Es mejor entonces no utilizar esta opción con las tablas.

Las funciones de respuesta del BDE

La última de las técnicas que exploraremos en este capítulo es el uso de funciones de respuestas del BDE. El API del Motor de Datos nos permite registrar algunas funciones de nuestra aplicación para que sean llamadas por el propio Motor durante algunas operaciones. De registrar la función de respuesta se encarga la siguiente rutina:

```
Word __stdcall DbiRegisterCallBack(hDBCiCur hCursor, CbType ecbType,
    int iClientData, Word iCbBufLen, void* CbBuf,
    pfDBICallBack pfCb);
```

El parámetro principal es *ecbType*, que indica qué operación deseamos controlar. Aunque hay unos cuantos más, he aquí algunos de sus posibles valores:

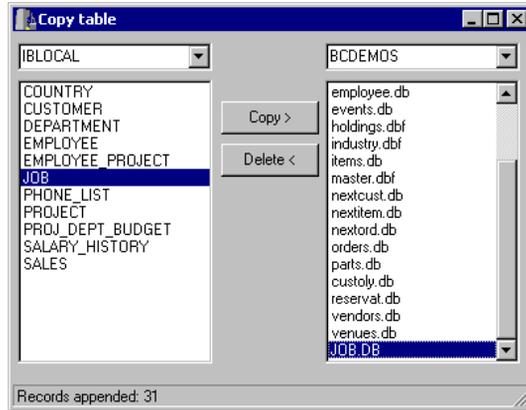
- *cbTABLECHANGED*: Es la que más fantasías despierta en el programador; vanas fantasías, precisaría yo. La función de respuesta se dispara cuando la instancia activa detecta un cambio en los datos de una tabla de Paradox. Esta es su principal limitación: no puede utilizarse con dBase, Access ni con servidores SQL.
- *cbCANCELQRY*: Seguimos con las funciones fantasiosas. En este caso, podemos detener la ejecución de una consulta ... pero solamente si el servidor es Sybase.
- *cbDELAYEDUPD*: Menciono esta constante para que el lector vea cómo la VCL ya intercepta algunas de las funciones de respuesta. En este caso, se trata de la función que dispara el evento *OnUpdateError* cuando hay actualizaciones en caché.
- *cbRESTRUCTURE*: ¿Ha visto los mensajes de advertencia que lanza Database Desktop cuando modificamos la estructura de una tabla? La función que se registra mediante esta constante es la responsable de esos mensajes.
- *cbGENPROGRESS*: Notifica periódicamente acerca del progreso de operaciones largas, en particular, de los movimientos de registros masivos mediante *DbiBatchMove* y, por extensión, del componente *TBatchMove*.

En cualquier caso, el prototipo de la función de respuesta debe corresponder al siguiente:

```
typedef CbRType __stdcall (*pfDBICallBack)
    (CbType ecbType, int iClientData, void *CbInfo);
```

Los parámetros *CbBuf* y *iCbBufLen* de *DbiRegisterCallBack* deben suministrar un *buffer* cuya estructura depende del tipo de función de respuesta.

El ejemplo que le voy a mostrar registrará una función de respuesta para la operación *DbiBatchMove*, que nos informe acerca del progreso de la misma. La siguiente imagen muestra a la aplicación, que puede encontrar íntegramente en el CD-ROM, en pleno funcionamiento:



Cuando el tipo de función es *cbGENPROGRESS*, el tipo del *buffer* debe ser el siguiente:

```
struct CBPROGRESSDesc {
    short iPercentDone;
    char szMsg[128];
};
```

Las reglas del juego son las siguientes:

- Si *iPercentDone* es igual o mayor que cero, en este atributo se encuentra el porcentaje terminado de la operación.
- En caso contrario, *szMsg* contiene una cadena de caracteres indicando la fase de la operación y un valor numérico que indica el progreso realizado.

Es fácil entonces definir una función de respuesta de acuerdo a las reglas anteriores:

```
CBType __stdcall ProgressCB(CBType ecbType, int ClientData,
    void *buf)
{
    AnsiString msg = pCBPROGRESSDesc(buf)->iPercentDone == -1 ?
        pCBPROGRESSDesc(buf)->szMsg :
        IntToStr(pCBPROGRESSDesc(buf)->iPercentDone) + "%";
    wndMain->StatusBar1->SimpleText = msg;
    return cbrCONTINUE;
}
```

También podíamos haber devuelto *cb:ABORT*, para detener la copia de la tabla. Para simplificar, la función anterior se registra y se elimina del registro localmente, dentro una función auxiliar de la aplicación, a la cual hemos llamado *DoBatchMove*:

```
void __fastcall TwndMain::DoBatchMove()
{
    CBPROGRESSDesc prgDesc;
    Check(DbiRegisterCallBack(NULL, cbGENPROGRESS, 0,
        sizeof(prgDesc), (void*) &prgDesc, ProgressCB));
    try {
        BatchMove1->Execute();
    }
    __finally {
        Check(DbiRegisterCallBack(NULL, cbGENPROGRESS, 0, 0, 0, 0));
    }
}
```

Creación de instalaciones

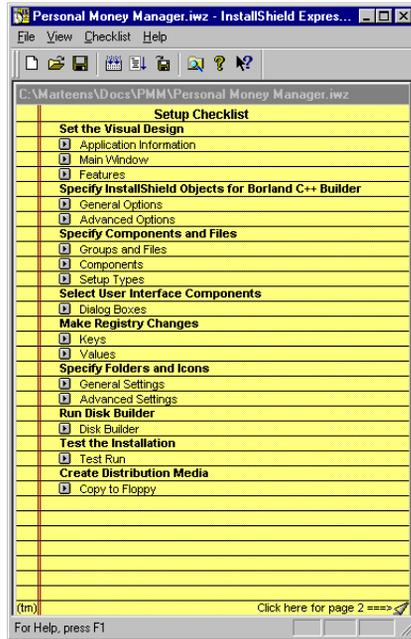
EL TOQUE FINAL DEL DESARROLLO de una aplicación es la generación de un programa para su instalación. Para poder distribuir aplicaciones de bases de datos creadas mediante C++ Builder necesitamos distribuir también el Motor de Datos de Borland, y ello implica la necesidad de configurar este motor de la forma más automática posible. También podemos querer distribuir la base de datos inicial con la que trabaja la aplicación, más los ficheros de ayuda y la documentación en línea. A partir de la versión 3 también se hace necesario distribuir los paquetes, para las aplicaciones que hacen uso de ellos, lo que añade más complejidad a una instalación típica.

Borland/Inprise incluye con C++ Builder una versión reducida del programa de creación de instalaciones InstallShield. A continuación estudiaremos cómo crear instaladores con esta aplicación. Al final del capítulo mencionaré las posibilidades de la versión completa de este producto.

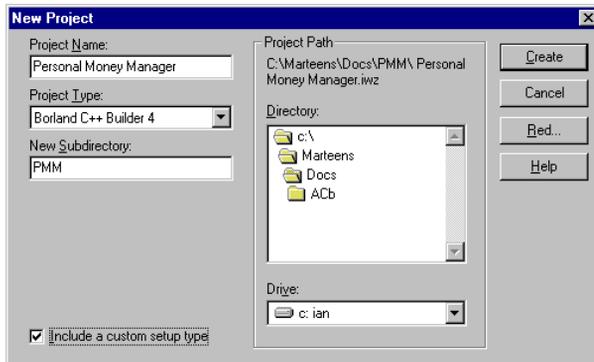
Los proyectos de InstallShield Express

InstallShield no se instala automáticamente junto con C++ Builder, y se debe ejecutar la opción correspondiente del programa de instalación global, que aparece al insertar el CD-ROM en el ordenador. Una vez que el programa se ha copiado en el ordenador, podemos acceder al mismo por medio de un acceso directo que se coloca directamente en el menú *Inicio | Programas*, de Windows.

InstallShield Express nos permite generar las instalaciones mediante una serie de diálogos de configuración en los cuales debemos indicar qué ficheros debemos copiar, qué parámetros de Windows debemos cambiar y qué interacción debe tener el programa de instalación con el usuario. Hay que aclarar este punto, porque existen versiones de InstallShield en las cuales las instalaciones se crean compilando un *script* desarrollado en un lenguaje de programación al estilo C. Evidentemente, el trabajo con estas versiones es mucho más complicado.



Para comenzar a trabajar con InstallShield necesitamos crear un *proyecto*, que contendrá los datos que se suministrarán al generador. Los proyectos se almacenan en ficheros de extensión *iwz*. Cuando se crea un proyecto nuevo, con el comando de menú *File|New*, hay que indicar el nombre y el directorio donde queremos situar este fichero. Si el directorio no existe, podemos crearlo tecleando su nombre en el cuadro de edición *New subdirectory*. También podemos abrir un proyecto existente, con el comando *File|Open*, o seleccionándolo de la lista de los últimos proyectos cargados, del menú *File*.

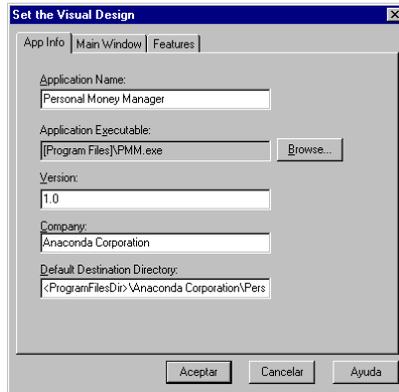


En el cuadro de diálogo de creación de proyectos hay una casilla importante, con el título *Include a custom setup type*. Debemos marcar esta casilla si queremos que el usuario de la instalación pueda seleccionar los componentes de la aplicación que quiere

instalar y los que no. Si no incluimos esta posibilidad al crear el proyecto, se puede indicar más adelante, pero nos costará más trabajo hacerlo.

La presentación de la instalación

Una vez creado el fichero de proyecto, los primeros datos que InstallShield necesita son los datos generales de la aplicación y los relacionados con la presentación del programa de instalación. Los datos de la aplicación se suministran en la sección *Application information*, en la que se nos piden los siguientes datos:



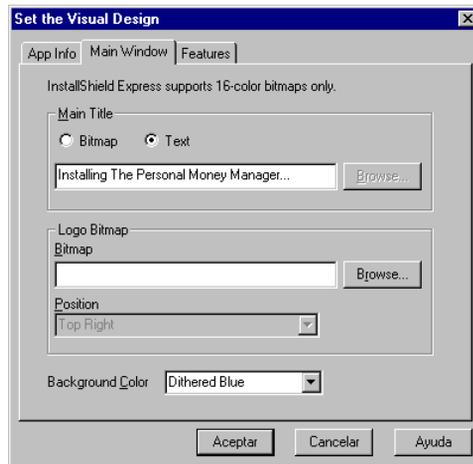
Dato	Observaciones
<i>Application Name</i>	El nombre de la aplicación. Puede ser diferente del nombre del ejecutable.
<i>Application Executable</i>	El nombre del fichero y su ubicación en el ordenador en que estamos trabajando.
<i>Version</i>	El número de versión de nuestra aplicación.
<i>Company</i>	El nombre de nuestra compañía.
<i>Default Destination Directory</i>	El directorio de instalación por omisión.

Los datos sobre el nombre de la aplicación, la empresa y la versión se utilizan para deducir otros parámetros de la instalación. Por ejemplo, el nombre del directorio de instalación inicial se construye a partir del directorio de programas de Windows, más el nombre de la empresa seguido por el nombre de la aplicación. Los datos de la aplicación, por otra parte, se almacenan en una clave del registro de configuración de Windows que se construye de forma similar, incluyendo además la versión. De esto hablaremos más adelante.

En la página *Main Window*, del mismo cuadro de diálogo, se ajustan las propiedades visuales de la ventana de presentación del instalador. Las propiedades configurables son tres: un texto, que aparece en cursivas en la esquina superior izquierda de la ven-

tana, un logotipo, del que podemos indicar su posición, y el color de fondo de la ventana. El texto de la presentación es, casi siempre, el mismo nombre que le hemos dado a la aplicación (*Application Name*), y recomiendo dejar el fondo de la ventana con el típico gradiente azul, a no ser que su aplicación se ocupe de la historia de los enanitos verdes, de Eric El Rojo, o del Submarino Amarillo de los Beatles.

En la opción *Logo Bitmap* podemos indicar la ubicación de un fichero de imagen, que puede ser un mapa de bits o un metafichero (extensiones *bmp* y *wmf*). Los gráficos que podemos utilizar con la versión de InstallShield para C++ Builder están limitados a 16 colores. Realmente, el código necesario para mostrar mapas de bits de 256 colores es un poco complicado, pues hay que tener en cuenta la existencia de paletas asociadas a los gráficos, y la posibilidad de que dos gráficos diferentes, con paletas disímiles, estén en pantalla al mismo tiempo. Tenemos una foto de un coche y una foto de una chica. La primera imagen utiliza una paleta en la que abundan los tonos metálicos mientras que en la segunda predominan suaves matices de rosa, marrón, amarillo y azul. En un sistema limitado a 256 colores, cuando ambas imágenes se visualicen simultáneamente sucederá que la chica parecerá un coche recién pintado, o que el coche mostrará unos sospechosos tonos carnales.



Por último, en la página *Features* tenemos una casilla para indicar si queremos la desinstalación automática de nuestra aplicación. No conozco motivo alguno para desactivar esta casilla.

Las macros de directorios

En muchas de las opciones que veremos a continuación que se refieren a directorios del ordenador donde se instala la aplicación, se pueden utilizar *macros* de directorios. Estas son nombres simbólicos, o variables de cadenas, y nos evitan las dependencias

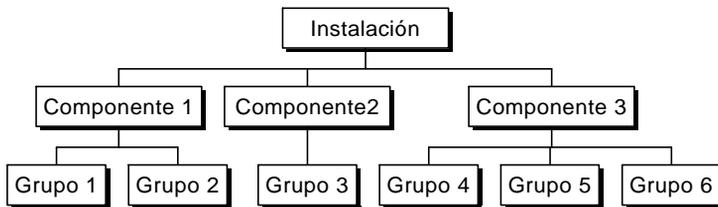
que causan los nombres absolutos. Por ejemplo, diferentes ordenadores pueden tener instalado Windows en directorios diferentes. Para no tener que teclear *c:\windows* en una opción y hacer dependiente la instalación de este directorio, se puede utilizar la macro `<WINDIR>`, que al ejecutarse la instalación es sustituida por el nombre del directorio donde reside el sistema operativo en el ordenador de destino.

Las macros de directorios de InstallShield Express son las siguientes:

Macro	Significado
<code><INSTALLDIR></code>	El directorio de instalación, suministrado por el usuario.
<code><WINDIR></code>	El directorio principal de Windows.
<code><WINSYSDIR></code>	El directorio de sistema de Windows.
<code><WINDISK></code>	El disco del directorio principal de Windows.
<code><WINSYSDISK></code>	El disco del directorio de sistema de Windows.
<code><WINSYS16DIR></code>	El directorio donde residen las DLLs de 16 bits.
<code><ProgramFilesDir></code>	El directorio de los archivos de programa.
<code><CommonFilesDir></code>	El directorio de los archivos comunes.

Grupos y componentes

La parte más importante de la especificación de un proyecto de InstallShield se configura en la sección *Specify Components and Files*. Aquí es donde se indican los ficheros que se copian durante la instalación y en qué directorios del ordenador de destino se deben instalar. También se pueden agrupar los ficheros en grupos y componentes, de modo que el usuario pueda escoger cuáles de estos grupos deben ser instalados y cuáles no. InstallShield agrupa el conjunto de ficheros a instalar según el siguiente esquema:

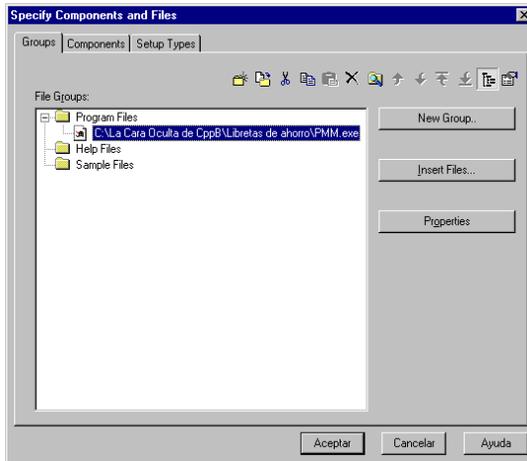


Un grupo está formado por una serie de ficheros que se instalan todos en el mismo directorio, mientras que un componente es una colección de grupos. Por ejemplo una aplicación determinada puede estar constituida por dos componentes: los ficheros de la aplicación y los ficheros de soporte. Los ficheros de la aplicación pueden estar divididos en tres grupos: los ejecutables, propiamente hablando, que residirían en el directorio raíz de la instalación, los ficheros de ayuda, que podrían ubicarse en un subdirectorio llamado *ayuda*, y las DLLs necesarias, que se copiarían al directorio de sistema de Windows del ordenador. Por su parte, los ficheros de soporte pueden

repartirse a su vez en grupos: el código fuente del programa en un directorio, documentación adicional en otro, etc.

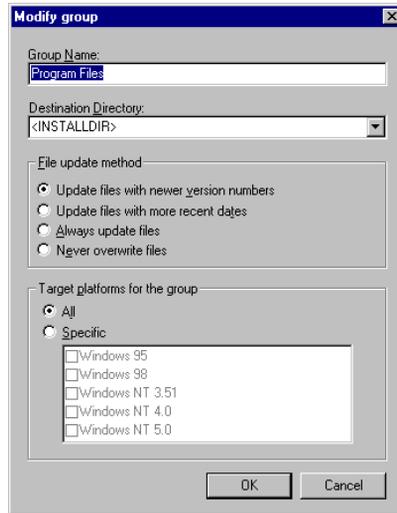
Los grupos y componentes iniciales de un proyecto dependen de la forma en que éste ha sido creado, de acuerdo al estado de la casilla *Include a custom setup type* del diálogo de creación de proyectos. Si no se permiten instalaciones personalizadas, se crea un único grupo, *Program files*, cuyos ficheros se copian al directorio de instalación. Este grupo forma parte del único componente, *Application files*. Por el contrario, cuando existe la opción de instalar componentes de la aplicación a la medida, se crean tres grupos de ficheros, a modo de orientación: *Program files*, *Help files*, y *Sample files*. No es necesario dividir nuestros ficheros en estos tres grupos, pero para muchas aplicaciones el esquema es válido. Se crean tres componentes a la vez, cada uno con uno de los grupos anteriores: *Application Files*, *Help and Tutorial Files* y *Sample Files*. El único grupo que no está vacío es *Program files*, que contiene el ejecutable de la aplicación, que se ha especificado en la sección *Application information*.

El único método mediante el cual se añadían ficheros a un grupo en algunas versiones de InstallShield era cuando menos curioso: había que arrastrar los ficheros desde el Explorador de Windows hasta la carpeta que representa al grupo. Era divertido observar a programadores que no tenían mucha habilidad con el ratón tratando de hacer diana en el minúsculo icono del grupo, o forcejeando con las ventanas de InstallShield y del Explorador de modo que ambas fueran visibles simultáneamente. En la versión que acompaña a C++ Builder 4, sigue existiendo un botón para ejecutar el Explorador de Windows:



Ahora también tenemos un botón *Insert files* para añadir ficheros a un grupo de una forma menos arriesgada, aunque sigue conservándose la posibilidad de arrastrar y soltar ficheros. Del mismo modo, se ha añadido la posibilidad de poder especificar

qué sucede cuando la instalación detecta que ya existen los ficheros que se quieren copiar al directorio del grupo:

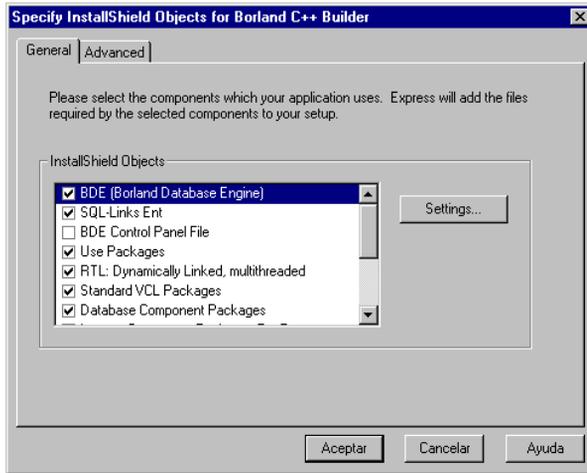


Podemos indicar que los ficheros se actualicen cuando tengan un número de versión más antiguo, o según la fecha, o que se actualicen siempre o nunca. También podemos especificar que un grupo se instale solamente para determinados sistemas operativos.

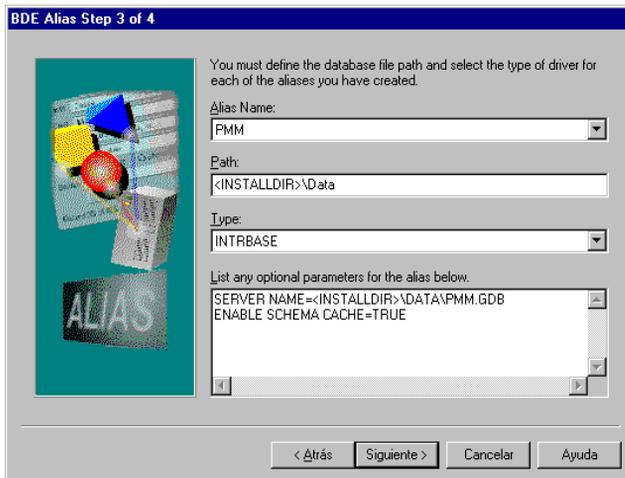
Una vez que se han definido los componentes y grupos de ficheros de una aplicación, debemos configurar los tipos de instalaciones. La versión *Express* de InstallShield solamente permite tres tipos de instalaciones: *Typical*, *Compact* y *Custom*. En el primer tipo se incluyen los componentes de uso más frecuente, en el segundo tipo de instalación se incluyen los componentes mínimos que necesita la aplicación, mientras que el tipo personalizado incluye inicialmente a todos los componentes, para que el usuario pueda posteriormente configurar la instalación a su gusto.

Instalando el BDE y los SQL Links

InstallShield nos permite utilizar la sección *General Options* para incluir dentro de la instalación los subcomponentes más usuales de C++ Builder. Estos son, en primer lugar, el Motor de Datos y los SQL Links. Pero también aprovecharemos dicha sección para incluir los paquetes predefinidos y la biblioteca de tiempo de ejecución, cuando queremos situarla en una DLL:



Cuando el programador escoge instalar el BDE, se ejecutan automáticamente los diálogos de configuración de la instalación del motor de datos. El primero de estos diálogos nos permite elegir entre una instalación completa o parcial. Es recomendable efectuar siempre una instalación completa, pues una parcial puede estropear una copia ya instalada del BDE. En el segundo diálogo podemos introducir los nombres de los alias persistentes que queremos crear, mientras que en el tercero indicamos si deseamos que la configuración nueva esté disponible tanto para las aplicaciones de 16 bits como para las de 32. Por último, por cada alias persistente a crear tenemos que suministrar su configuración: su controlador y sus parámetros, al igual que cuando creamos alias mediante el BDE. En el parámetro *PATH*, y en la lista de parámetros opcionales, se pueden utilizar las macros de directorio.



Aunque un alias SQL no necesita el parámetro *PATH* hace falta especificar “algo” para este valor, pues en caso contrario InstallShield no puede crear el alias. Utilice, por ejemplo, la macro de directorio *<INSTALLDIR>*.

Por su parte, la instalación de los SQL Links nos deja seleccionar con qué formatos deseamos trabajar. En este caso, no hay problemas por no realizar una instalación completa.

En la sección *Advanced Options* podemos ver qué ficheros y grupos se van a instalar con la aplicación. Este diálogo no permite realizar modificaciones, pues es solamente informativo. Si el programador hace posibles las instalaciones personalizadas, los grupos que se crean en *General options* no se añaden automáticamente a los componentes creados. Hay que regresar a la sección *Components* para incluir los nuevos grupos dentro de alguno de los componentes existentes.

Configuración adicional del BDE

Lamentablemente, con InstallShield solamente podemos configurar alias para el BDE, mientras que nuestras instalaciones pueden necesitar otros cambios en la configuración global y de los controladores del Motor de Datos. Por ejemplo, las aplicaciones diseñadas para Paradox necesitan configurar los parámetros *NET DIR* y *LOCAL SHARE* para su correcto funcionamiento.

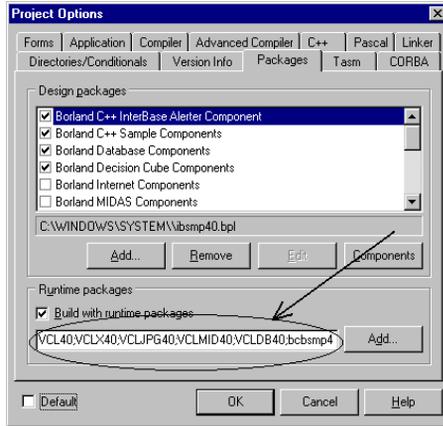
Al final de este capítulo veremos cómo programar DLLs que puedan ejecutarse desde la versión Express de InstallShield (no es la que viene con C++ Builder). En una de estas extensiones podemos añadir código para configurar el BDE. Si no utilizamos la versión Express, estamos obligados a realizar la configuración de los parámetros de controladores desde nuestra propia aplicación, quizás la primera vez que se inicie. En el capítulo anterior hemos visto cómo cambiar dichos parámetros.

Instalación de paquetes

Como hemos visto, es muy fácil instalar los paquetes predefinidos con la versión de InstallShield que acompaña a C++ Builder 4. Sin embargo, podemos vernos en la necesidad de instalar otros paquetes adicionales o estar obligados a seguir un tiempo con la versión 3.

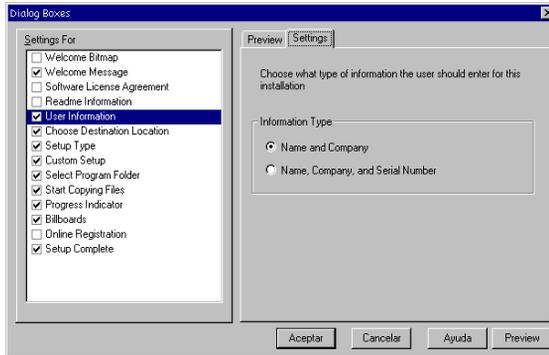
En esta situación, es aconsejable crear un grupo de ficheros y darle un nombre, digamos que *Paquetes*. En su propiedad *Destination Directory* debemos especificar la macro *<WINSYSDIR>*. A este grupo arrastramos los ficheros de paquetes de la versión de C++ Builder con la que estamos trabajando. ¿Qué ficheros son estos? Te-

remos que buscar los ficheros de extensión *bpl* que se encuentran en el directorio de sistema de Windows. Sin embargo, es más fácil determinar cuáles son exactamente los paquetes necesarios si abrimos el diálogo de las opciones de proyecto de C++ Builder, y nos fijamos en el contenido del cuadro de edición *Runtime packages*:



Interacción con el usuario

La sección *Dialog boxes* permite configurar la mayor parte de los cuadros de diálogos con los que el programa instalador realiza su interacción con el usuario final de nuestra aplicación. En esta sección existen entradas para cada uno de los diálogos típicos de una instalación. En la mayoría de los casos, la configuración consiste en decir si vamos a mostrar o no el diálogo en cuestión; para algunos diálogos, tenemos la posibilidad de configurar los elementos que lo constituyen, o especificar valores iniciales para los datos con que trabajan. Siempre se puede, además, tener una idea del aspecto final del diálogo pulsando el botón *Preview*.



A continuación describiremos los elementos de presentación de InstallShield:

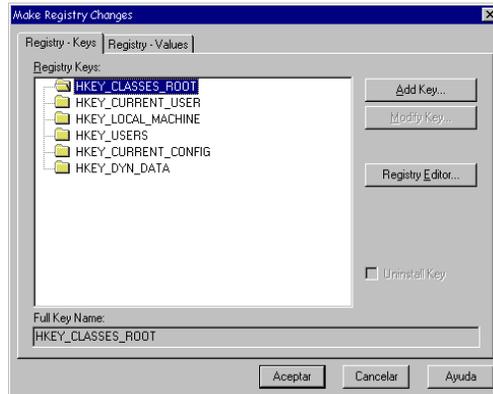
- *Welcome Bitmap*: la imagen de bienvenida.
No se muestra por omisión. Si se activa este diálogo, hay que especificar un mapa de bits de 16 colores en la página *Settings*.
- *Welcome Message*: el mensaje de bienvenida.
El contenido de este cuadro de diálogo no se puede cambiar.
- *Software License Agreement*: el acuerdo de licencia de la aplicación.
No se muestra por omisión. Cuando se activa, el texto a mostrar se extrae de un fichero que se especifica en la página *Settings*. El diálogo tiene un botón *No*, por si el usuario decide no aceptar los términos de la licencia.
- *Readme Information*: información antes de instalar.
No se muestra por omisión. Cuando se activa, el texto a mostrar se extrae de un fichero que se especifica en la página *Settings*.
- *User Information*: información acerca del usuario.
Este diálogo pide el nombre del usuario y la compañía a la que pertenece. Se inicializa con los valores almacenados por Windows en su registro, y los valores suministrados se almacenan en una clave dedicada a la aplicación, como explicaré más adelante. En la página *Settings* puede indicarse si queremos, además, el número de serie del programa.
- *Choose Destination Location*: ubicación final de la aplicación.
Permite especificar el directorio de instalación. En la página *Settings* puede cambiarse el directorio por omisión.
- *Setup Type*: tipo de instalación.
Si queremos que el usuario pueda personalizar su instalación, debemos activar esta opción. Si al crear el proyecto, decidimos incluir esta posibilidad, la casilla correspondiente ya estará activada.
- *Custom Setup*: instalaciones personalizadas.
Esta casilla se activa y se desactiva en sincronía con la del diálogo anterior; se incluye sólo por compatibilidad con otras versiones de InstallShield. El diálogo que aparece permite al usuario escoger qué componentes desea instalar si ha elegido una instalación personalizada. También permite cambiar el directorio de instalación.
- *Select Program Folder*: seleccionar la carpeta.
Permite cambiar el nombre de la carpeta donde se colocan los iconos del programa, o elegir una carpeta existente. En la página *Settings* se puede indicar el nombre inicial de la carpeta.
- *Start Copying Files*: mensaje de inicio de la copia.
Este diálogo no es configurable, y muestra los valores que se utilizarán para la instalación, antes de llevarla a cabo.
- *Progress Indicator*: indicador de progreso.
Es el diálogo que muestra el porcentaje efectuado de la instalación. No es configurable.

- *Billboards*: carteles.
Mientras InstallShield copia los ficheros en el ordenador, se pueden mostrar imágenes para información o entretenimiento del usuario. En la pagina *Settings* hay que teclear el nombre del directorio donde se encuentran las imágenes, que deben ser mapas de bits o metaficheros de 16 colores (extensiones *bmp* y *wmf*). Los ficheros de imágenes deben, por convenio, nombrarse *setup1.bmp* (ó *wmf*), *setup2.bmp*, etc. InstallShield determina automáticamente los intervalos entre imágenes.
- *Online Registration*: registro en línea.
- *Setup Complete*: instalación terminada.
Con este diálogo, permitimos que el usuario pueda lanzar uno de nuestros ejecutables al finalizar la aplicación, o que pueda reiniciar el ordenador.

Las claves del registro de Windows

El registro de Windows almacena los datos de configuración de los programas instalados en el sistema, y sustituye a la multitud de ficheros *ini* necesarios en versiones anteriores del sistema operativo. Podemos crear e inicializar secciones y valores en el registro mediante la sección *Make Registry Changes*. Antes de ver cómo podemos añadir claves y valores debemos conocer las claves que InstallShield llena automáticamente en el registro. La más importante de todas es la cadena de desinstalación, que se guarda en la clave siguiente:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\  
Current Version\Uninstall\<Aplicación>
```



Bajo esta clave se almacenan las cadenas *DisplayName* y *UninstallString*. La primera representa el nombre que se muestra en el Panel de Control, en el comando de agregar o quitar programas. La segunda es el comando que hay que ejecutar para eliminar

la aplicación del sistema. El nombre de la aplicación es, por supuesto, el suministrado por el programador en el apartado *Application name*.

Antes de Windows 95, los programas para MS-DOS y Windows 3.x necesitaban modificar el fichero *autoexec.bat* para añadir directorios a la variable *path*. Esto ya no es necesario, pues Windows 95 ofrece una clave especial en el registro, bajo la cual la aplicación que lo desee puede guardar su directorio de ejecución:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\
  Current Version\App Paths\Aplicación
```

Por otra parte, si el programador solicita información sobre los datos del usuario, en la sección *Dialog Boxes*, con la opción *User information*, InstallShield almacena los datos que suministra el usuario bajo la clave:

```
HKEY_LOCAL_MACHINE\SOFTWARE\<Compañía>\<Aplicación>\<Versión>
```

Bajo la clave anterior se almacenan las cadenas *User*, *Company* y *Serial*; esta última si ha sido especificada en la configuración de *User information*.

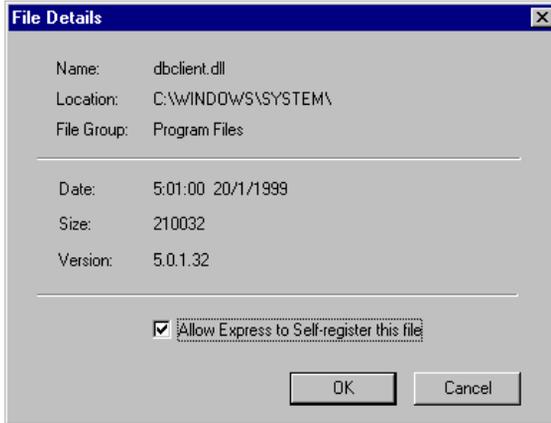
Si nuestra aplicación necesita crear claves y valores adicionales, debemos utilizar la sección *Make Registry Changes*. En la primera página se crean las claves; si la clave no pertenece a las predefinidas por InstallShield, es necesario marcar la casilla *Uninstall key*, para que la clave se borre al desinstalar el programa. Para asignar valores dentro de claves nuevas o ya existentes, utilice la segunda página del cuadro de diálogo.

Cómo se registran los componentes ActiveX

Uno de los problemas más frecuentes que plantea una instalación es registrar los servidores COM que necesita la aplicación a instalar, en particular, los controles ActiveX. Como vimos en el capítulo 34, todos los servidores COM son capaces de registrarse a sí mismos automáticamente. La pregunta es: ¿cómo puede saber InstallShield si determinado ejecutable o DLL necesita ser registrado para su correcto funcionamiento?

InstallShield busca dentro de la información de versión del fichero una cadena con el contenido *OleSelfRegister*. Si la encuentra, el fichero se registra. Recuerde que si se trata de una DLL, debe ejecutarse la función exportada *DllRegisterServer*, y que si es un ejecutable, debe lanzarse con el parámetro */regserver*.

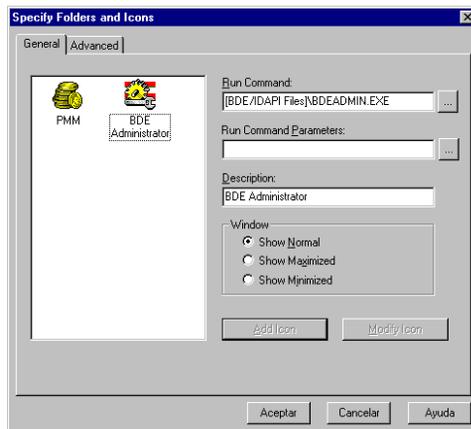
Ahora bien, no todos los servidores COM incluyen dicha cadena en la información de versión. Para estos casos, la versión de InstallShield que viene con C++ Builder 4 permite indicar, en las propiedades del fichero que se activan en la ventana de grupos y componentes, si queremos o no registrarlos:



Si no dispone de esta versión, puede intentar registrar manualmente el servidor al inicio de su aplicación. Entre los ejemplos y utilidades de C++ Builder se encuentra un programa llamado *regsvr*, que muestra cómo registrar ejecutables y DLLs.

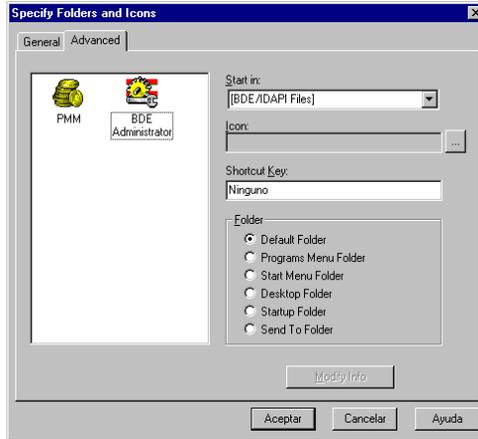
Iconos y carpetas

Para que el usuario final pueda acceder a la aplicación recién instalada, InstallShield crea iconos y carpetas que se incluyen en el menú de inicio de Windows 95 y NT 4, o en el Administrador de Programas de WinNT 3.5x. Por omisión, InstallShield crea un icono para la aplicación especificada en *Application name*, pero se pueden crear todos los iconos necesarios y configurarlos en las secciones *General settings* y *Advanced settings* de *Specify Folders and Icons*.



Para añadir un nuevo icono, teclee el nombre del comando en la primera página del diálogo, en el apartado *Run command*, o seleccione uno de los ficheros de la instala-

ción. En *Run command parameters* hay que indicar los parámetros que se le suministran a la aplicación asociada al icono. Por último, en *Description* se configura el texto asociado al icono. La versión que acompaña a C++ Builder 4 permite además indicar el modo de visualización inicial de la aplicación: normal, a pantalla completa o minimizada.



En la segunda página se completan los parámetros de cada icono. Estos son el directorio de trabajo (*Start in*), el icono (*Icon*), y la tecla de abreviatura (*Shortcut key*). También puede indicarse, mediante el grupo de selección *Folder*, en qué carpeta queremos situar el acceso directo. Por ejemplo, si activamos la opción *Start Menu Folder* el icono se situará directamente en el primer nivel del menú *Programas* de Windows. Esto es recomendable solamente cuando la aplicación instala un solo icono, pues en caso contrario se congestiona demasiado este menú desplegable.

Generando y probando la instalación

Cuando todos los parámetros de la instalación han sido suministrados, es hora de crear los discos de la instalación, utilizando la opción *Disk Builder*. Podemos, antes de generar la instalación, indicar el formato en que queremos los discos. InstallShield permite utilizar los siguientes formatos:

Opción	Formato
720KB	Discos de 3.5 pulgadas, simple densidad
1.2MB	Discos de 5.25 pulgadas, doble densidad
1.44MB	Discos de 3.5 pulgadas, doble densidad
2.88MB	Discos de 3.5 pulgadas, cuádruple densidad
120MB	Para discos de mayor capacidad
CD-ROM	Hasta 650MB

En cualquier caso, la escritura no se efectúa directamente sobre el medio seleccionado. El generador de instalaciones crea un subdirectorio bajo el directorio del proyecto, utilizando el nombre del formato. Por ejemplo, si seleccionamos discos de doble densidad de 3.5 pulgadas, el subdirectorio se llamará *144mb*. Bajo este subdirectorio, a su vez, se crearán dinámicamente otros subdirectorios llamados *Disk1*, *Disk2*, etc., con el contenido de cada uno de los discos de instalación. Una vez elegido el formato de los discos, pulsamos el botón *Build* para generar sus contenidos. Se puede utilizar este comando varias veces, con formatos diferentes, si necesitamos distribuir la aplicación en soportes de diferente tamaño.

Finalmente, puede probar la instalación desde la opción *Test Run*. Tenga en cuenta que el programa se instalará realmente en su ordenador, pues no se trata de simular la instalación.

La versión completa de InstallShield Express

InstallShield Express 2, la versión completa del producto que se puede comprar por separado, ofrece características interesantes para el programador. Estas son algunas de ellas:

- Se incluyen módulos adicionales de lenguajes, para cambiar el lenguaje de la instalación. Si quiere esta posibilidad, asegúrese que se trata de la versión 2. La primera versión de InstallShield Express solamente desarrolla instalaciones en inglés.
- Se pueden desarrollar instalaciones para plataformas de 16 y 32 bits.
- Los sistemas de desarrollo soportados son varios: Delphi, Borland C++ y C++ Builder, Visual Basic, Optima++, etc.
- Es posible ejecutar *extensiones* de InstallShield desde la instalación. Estas son DLLs o ejecutables desarrollados por el programador que realizan acciones imposibles de efectuar por InstallShield.
- Se pueden modificar los ficheros *autoexec.bat* y *config.sys*. Esto es indispensable en instalaciones para 16 bits.
- Se pueden mezclar ficheros de extensión *reg* en el Registro. Esta técnica permite registrar fácilmente los controles ActiveX.
- Los accesos directos y entradas en el menú que se crean para las aplicaciones tienen más opciones de configuración. Podemos, por ejemplo, indicar que cierta aplicación se ejecute minimizada o maximizada, y colocar una aplicación en otra carpeta, como la de aplicaciones de inicio.
- Se pueden indicar ficheros temporales a la instalación, que se eliminan una vez finalizada la misma, dejar espacio en el primer disco para ficheros que se instalan sin descomprimir, y crear instalaciones autoejecutables desde un solo fichero *exe*.

En mi opinión, la compra de InstallShield Express 2 es una inversión necesaria y rentable para el programador profesional de C++ Builder.

Las extensiones de InstallShield Express

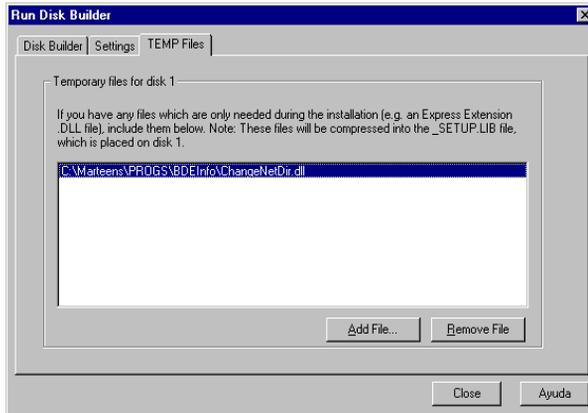
InstallShield Express permite llamar a ficheros ejecutables y a funciones situadas dentro de DLLs desde una instalación. Desde estos módulos podemos realizar tareas tales como terminar la configuración del BDE, pidiendo quizás información al usuario. Aquí mostraremos un pequeño ejemplo de cómo modificar el parámetro *NET DIR* al concluir una instalación. Por simplicidad, utilizaré un directorio predeterminado para dicho parámetro. El lector puede completar el ejemplo incluyendo un diálogo que permita al usuario seleccionar un directorio.

Creamos una DLL con C++ Builder, e incluimos en la misma la unidad *BDEChange* que hemos mostrado en el capítulo anterior. Llamaremos a la DLL *ChangeNetDir*, y crearemos en su fichero principal la siguiente función:

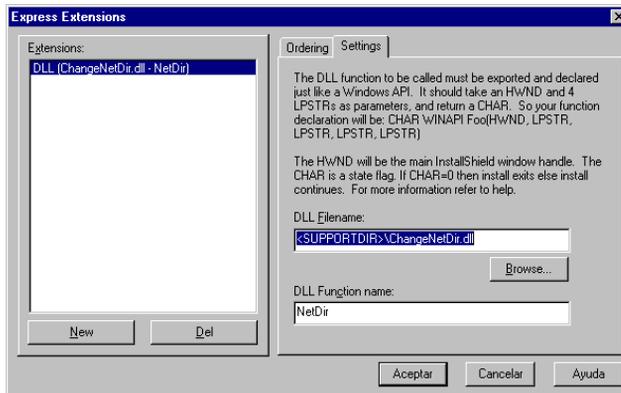
```
extern "C" {
__declspec(dllexport) char WINAPI NetDir(HWND Hwnd,
    LPSTR Source, LPSTR Support, LPSTR Install, LPSTR)
{
    DbiInit(0);
    try { SetNetDir("C:\Windows\Temp"); }
    __finally { DbiExit(); }
    return '\x01'
}
}
```

El prototipo de *NetDir* es el exigido por InstallShield Express para que una función externa pueda ser llamada. El parámetro *Hwnd* indica la ventana que debe utilizarse como ventana madre si queremos lanzar algún cuadro de diálogo desde la DLL. *Source*, *Support* e *Install* son los nombres de los directorios de InstallShield, que no utilizaremos en el ejemplo. La función debe retornar un carácter. Como convenio, si retornamos el carácter nulo se aborta la instalación. Cualquier otro carácter es ignorado. Por supuesto, esta función debe ser exportada por la biblioteca.

Una vez compilada la DLL, debemos incluirla dentro de la instalación. Si se trata de una DLL temporal, que no será utilizada por la aplicación más adelante, lo mejor es comprimirla con el resto de los ficheros temporales de InstallShield. Esto se especifica en el apartado *Disk builder* de la versión Express:



Más adelante, en el apartado *Express extensions*, creamos una nueva extensión, indicando que se trata de una DLL. Tenemos que indicar en qué momento de la instalación se ejecutará la extensión. Lo más indicado es que se ejecute al final, cuando ya se ha instalado el BDE, que la extensión necesita. En la página *Settings* del diálogo de configuración de extensiones debemos indicar el nombre de la DLL y el directorio donde InstallShield debe buscarla. Como se trata de una DLL temporal, que hemos situado junto al núcleo de la instalación, debemos utilizar la macro de directorios `<SUPPORTDIR>`:



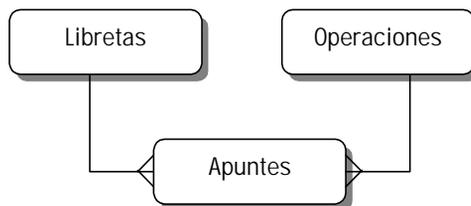
Y ya está lista la extensión para su ejecución durante la instalación de nuestra aplicación.

Ejemplos: libretas de ahorro

NO SE TRATA DE ASALTAR UN BANCO, aunque es probable que cuando termine esta aplicación le venga el deseo de entrar a saco en alguno²⁵. Me gustaría finalizar el libro con varios ejemplos sencillos que muestren como integrar muchos de los conceptos que hemos explicado. El ejemplo de este capítulo trata sobre la administración de un conjunto de libretas de ahorro, en las cuales se van realizando apuntes, mientras se mantiene actualizado el saldo de las mismas. El ejemplo se desarrollará inicialmente en InterBase, de modo que puedan seguirlos los usuarios de las versiones Professional y Enterprise, pero también mostraremos cómo adaptarlos a Oracle y a MS SQL Server 7.

Descripción del modelo de datos

El modelo de datos contendrá tres tablas: una para las libretas de ahorros, con los datos del titular, la descripción de la libreta, y con determinados valores consolidados auxiliares, como el saldo acumulado y el número de apuntes. Otra tabla servirá para la descripción de las operaciones en las libretas: descripción alfabética de la operación, código numérico y tipo de operación (si es un ingreso o un reintegro). La tercera tabla contendrá todos los apuntes realizados en todas las libretas, asociando apunte con libreta mediante un código de referencia a la libreta. Las relaciones de integridad referencial entre estas tres tablas son las siguientes:



Las tablas almacenarán la siguiente información:

²⁵ Las noticias de asaltos a bancos son en cierto sentido similares al conocido chiste de “Hombre muerde a perro”.

Tabla	Columna	Significado
<i>Libretas</i>	<i>Código</i>	Número que identifica a la libreta
	<i>Titular</i>	El titular de la libreta
	<i>Descripción</i>	Descripción de la libreta
<i>Operaciones</i>	<i>Código</i>	Un código numérico de operación
	<i>Descripción</i>	La descripción de la operación
	<i>Ingreso</i>	¿Es un ingreso o un reintegro?
<i>Apuntes</i>	<i>Libreta</i>	El código de la libreta a la que pertenece el apunte
	<i>Numero</i>	Número secuencial dentro de la libreta
	<i>Fecha</i>	La fecha del apunte
	<i>Operación</i>	El código de la operación
	<i>Importe</i>	El importe de la operación

Observe que, por simplificar, no hemos implementado la gestión de titulares como una entidad por separado. Tampoco hemos utilizado el convenio oficial de numeración de cuentas bancarias, a favor de un código artificial para las libretas. En cualquier caso, nunca se me hubiera ocurrido utilizar el número de cuenta como clave primaria por dos razones: es una clave compuesta y ocupa demasiado espacio. En el caso de haber incluido esa notación, seguiría utilizando la clave artificial como clave primaria.

Para facilitar la verificación de las reglas de consistencia y la extracción de información, añadiremos los siguientes campos redundantes al esquema:

Tabla	Columna	Significado
<i>Libretas</i>	<i>Apuntes</i>	Número de apuntes en la libreta
	<i>Saldo</i>	Saldo acumulado
	<i>Fecha</i>	Fecha del último apunte
<i>Apuntes</i>	<i>Saldo</i>	Saldo en el momento de la operación

Verificaremos el cumplimiento de algunas reglas, que detallamos a continuación:

- Las fechas de los apuntes de una misma libreta deben formar una secuencia monótona no decreciente. Esto debe verificarse durante la grabación de un nuevo apunte, pero también al corregir la fecha de un apunte existente.
- Solamente puede eliminarse el último apunte introducido en una libreta.
- Sólo puede corregirse el código de operación de un apunte si el nuevo código se refiere a una operación del mismo tipo (ingreso/reintegro).
- No se puede borrar una operación que tenga apuntes asociados.
- No se puede modificar el tipo de una operación con apuntes asociados.
- El usuario deberá poder teclear los importes de cada apunte sin preocuparse por su signo. La aplicación lo deducirá de acuerdo al tipo de operación.
- No se admitirán apuntes con importes de valor 0.

A continuación incluyo la parte del *script* que crea las tablas, índices y dominios para InterBase 5.5. He aprovechado la implementación de las acciones referenciales en las versiones de este sistema a partir de la 5. Si el lector desea utilizar alguna versión anterior, le bastará con eliminar las cláusulas **on delete** y **on update** de las restricciones de integridad referencial.

```

/* Creación de la base de datos */

create database "C:\La Cara Oculta de CppB\Datos\Bancos.GDB"
user "SYSDBA" password "masterkey";

/* Dominios */

create domain DCodigo as
integer not null;

create domain DDescripcion as
varchar(35) not null
check (value <> '');

create domain DBoolean as
char(1) not null
check (value in ('S', 's', 'N', 'n'));

create domain DMoney as
integer default 0 not null;

/* Tablas */

create table Operaciones (
Codigo          DCodigo,
Descripcion     DDescripcion,
Ingreso        DBoolean,

primary key     (Codigo),
unique         (Descripcion)
);

create table Libretas (
Codigo          DCodigo,
Titular         DDescripcion,
Descripcion     DDescripcion,
/* Columnas redundantes */
Saldo          DMoney,
Apuntes        int default 0,
Fecha          date,

primary key     (Codigo),
unique         (Titular, Descripcion)
);

create table Apuntes (
Codigo          DCodigo,
Libreta        DCodigo,
Numero         DCodigo,
Fecha          date not null,
Operacion      DCodigo,

```

```

    Importe      DMoney check (Importe <> 0),
    Saldo       DMoney,

    primary key (Codigo),
    unique      (Libreta, Numero),
    foreign key (Libreta) references Libretas(Codigo)
                on delete cascade on update cascade,
    foreign key (Operacion) references Operaciones(Codigo)
                on delete no action on update cascade
);

/* Indices */

create index FechaApunte on Apuntes(Fecha);

```

Debemos distinguir entre el código de un apunte (*Codigo*) y su número (*Numero*). El primero servirá como clave primaria de la tabla de apuntes, mientras que el segundo será un valor consecutivo dentro de cada libreta, pero que puede repetirse a nivel global. En realidad, hubiéramos podido designar al par libreta y número como la clave primaria de *Apuntes*, pero hubiera sido más complicado, fundamentalmente porque los números de apuntes van a ser asignados en el servidor, y el lector ya conoce las implicaciones negativas que tiene esta decisión desde el punto de vista de las herramientas clientes.

Este es el *script* que crea las excepciones, generadores, *triggers* y procedimientos almacenados:

```

connect "C:\La Cara Oculta de CppB\Datos\Bancos.GDB"
user "SYSDBA" password "masterkey";

/* Excepciones */

create exception FechaIncorrecta
    "El nuevo apunte es anterior a la fecha del último";
create exception ErrorCambioSigno
    "No se puede cambiar el signo a una operación en uso";
create exception CambioFechaIncorrecto
    "El cambio de fecha de apunte rompe la secuencia de fechas";
create exception CambioOperacionIncorrecto
    "La nueva operación no es del mismo tipo que la anterior";
create exception BorradoApunteIncorrecto
    "Sólo se puede eliminar el último apunte de una libreta";

/* Generadores */

create generator CodigoApunte;

/* Triggers y procedimientos almacenados */

set term !;

```

```

create procedure ObtenerCodigo returns (Codigo int) as
begin
    Codigo = gen_id(CodigoApunte, 1);
end!

create trigger TipoOperacion for Apuntes
    active before insert position 1 as
declare variable TipoOp char;
begin
    select Ingreso
    from Operaciones
    where Codigo = new.Operation
    into :TipoOp;
    if (new.Importe < 0) then
        new.Importe = - new.Importe;
    if (TipoOp in ('N', 'n')) then
        new.Importe = - new.Importe;
end!

create trigger MantenerSaldo for Apuntes
    active before insert position 2 as
declare variable Ultimo date;
begin
    /* Buscar el número de apunte y el saldo acumulado */
    select Saldo, Apuntes, Fecha
    from Libretas
    where Codigo = new.Libreta
    into new.Saldo, new.Numero, :Ultimo;
    /* Verificar la fecha del nuevo apunte */
    if (Ultimo is not null and Ultimo > new.Fecha) then
        exception FechaIncorrecta;
    /* Modificar valores a insertar */
    new.Saldo = new.Saldo + new.Importe;
    new.Numero = new.Numero + 1;
    /* Actualizar la libreta */
    update Libretas
    set Saldo = new.Saldo, Apuntes = new.Numero,
        Fecha = new.Fecha
    where Codigo = new.Libreta;
end!

create trigger ModApunte for Apuntes
    active before update position 1 as
declare variable Fec date;
declare variable TipoOp1 char(1);
declare variable TipoOp2 char(1);
begin
    /* Sólo se puede cambiar la fecha y la operación */
    if (old.Fecha <> new.Fecha) then
    begin
        select Fecha
        from Apuntes
        where Libreta = old.Libreta and
            Numero = old.Numero - 1
        into :Fec;
        if (Fec is not null and new.Fecha < Fec) then
            exception CambioFechaIncorrecto;
        select Fecha
        from Apuntes

```

```

        where Libreta = old.Libreta and
        Numero = old.Numero + 1
    into :Fec;
    if (Fec is not null and new.Fecha > Fec) then
        exception CambioFechaIncorrecto;
    if (Fec is null) then
        update Libretas
        set Fecha = new.Fecha
        whereCodigo = old.Libreta;
end
if (old Operacion <> new Operacion) then
begin
    select Ingreso
    from Operaciones
    whereCodigo = old Operacion
    into :TipoOp1;
    select Ingreso
    from Operaciones
    whereCodigo = new Operacion
    into :TipoOp2;
    if (TipoOp1 <> TipoOp2) then
        exception CambioOperacionIncorrecto;
    end
end!

create trigger EliminarApunte for Apuntes
active before delete position 1 as
declare variable Apu integer;
declare variable Fec date;
begin
    /* Sólo se permite si es el último apunte */
    select Apuntes
    from Libretas
    whereCodigo = old.Libreta
    into :Apu;
    if (Apu <> old.Numero) then
        exception BorradoApunteIncorrecto;
    /* Buscar la fecha del último apunte */
    select Fecha
    from Apuntes
    where Libreta = old.Libreta and
    Numero = old.Numero - 1
    into :Fec;
    /* Corregir los valores acumulados en la libreta */
    update Libretas
    set Apuntes = Apuntes - 1,
    Saldo = Saldo - old.Importe,
    Fecha = :Fec
    whereCodigo = old.Libreta;
end!

create trigger ModificarOperacion for Operaciones
active before update position 1 as
declare variable Cant integer;
begin
    /* Sólo se cambia el signo si no hay apuntes asociados */
    if (new.Ingreso <> old.Ingreso) then
    begin
        select count(*)
        from Apuntes

```

```

        where Operacion = old.Codigo
        into :Cant;
        if (Cant > 0) then
            exception ErrorCambioSigno;
        end
    end!

create procedure ModificarApunte(Cod int, Imp int) as
declare variable Lib int;
declare variable Num int;
declare variable ImpAnt int;
declare variable CodOp int;
declare variable TipoOp char(1);
begin
    /* Recuperar datos acerca del apunte a corregir */
    select Libreta, Numero, Importe, Operacion
    from Apuntes
    where Codigo = :Cod
    into :Lib, :Num, :ImpAnt, :CodOp;
    /* Corregir el signo del nuevo importe */
    select Ingreso
    from Operaciones
    where Codigo = :CodOp
    into :TipoOp;
    if (Imp < 0) then Imp = - Imp;
    if (TipoOp in ('N', 'n')) then Imp = - Imp;
    /* Corregir el apunte actual */
    update Apuntes
    set Importe = :Imp, Saldo = Saldo + :Imp - :ImpAnt
    where Codigo = :Cod;
    /* Propagar cambios en el saldo */
    update Apuntes
    set Saldo = Saldo + :Imp - :ImpAnt
    where Libreta = :Lib and
        Numero > :Num;
    /* Corregir resumen en la libreta */
    update Libretas
    set Saldo = Saldo + :Imp - :ImpAnt
    where Codigo = :Lib;
end!

set term ;!

```

Un poco largo, ¿verdad? El generador *CodigoApunte* y el procedimiento almacenado *ObtenerCodigo* serán utilizados por la aplicación para asignar automáticamente un código artificial a cada nuevo apunte. Hemos decidido no permitir directamente la modificación del importe de un apunte; hay que utilizar el procedimiento almacenado *ModificarApunte* para este propósito.

Libretas de ahorro en MS SQL Server

El siguiente listado contiene la definición de tablas, índices, procedimientos y *triggers* para Transact-SQL. Por comodidad, al principio del listado se crea y activa una base

de datos *Bancos*, pero también podemos utilizar las herramientas gráficas para la creación de la base de datos:

```
-- Creación y activación de la base de datos

create database Bancos
  on primary (
    name = BancosDat, filename =
      "C:\La Cara Oculta de CppB\Datos\BancosDat.mdf",
    size = 1MB, filegrowth = 10%)
  log on (
    name = BancosLog, filename =
      "C:\La Cara Oculta de CppB\Datos\BancosLog.ldf",
    size = 1MB, filegrowth = 10%)
go

sp_dboption 'Bancos', 'recursive triggers', 'FALSE'
go

use Bancos
go

-- Creación de tablas e índices

create table Operaciones (
  Codigo          int not null,
  Descripcion     varchar(30) not null,
  Ingreso        char not null default 'S',

  primary key    (Codigo),
  unique         (Descripcion),
  check          (Ingreso in ('S', 's', 'N', 'n'))
)

create table Libretas (
  Codigo          int not null,
  Titular         varchar(35) not null,
  Descripcion     varchar(35) not null,
  -- Campos redundantes
  Saldo          int not null default 0,
  Apuntes        int not null default 0,
  Fecha          datetime null default null,

  primary key    (Codigo),
  unique         (Titular, Descripcion)
)

create table Apuntes (
  Codigo          int not null,
  Libreta         int not null,
  Numero         int not null default -1,
  Fecha          datetime not null default (getdate()),
  Operacion      int not null,
  Importe        int not null,
  Saldo          int null default null,

  primary key    (Codigo),
  unique clustered (Libreta, Numero),
```

```

        foreign key      (Libreta) references Libretas(Codigo),
        foreign key      (Operacion) references Operaciones(Codigo)
    )

create index FechaApunte on Apuntes(Fecha)
create index FK_Apuntes_Operacion on Apuntes(Operacion)
go

-- Códigos automáticos para la tabla de apuntes

create table Codigos (
        Codigo          int not null
    )
insert into Codigos values (1)
go

create procedure ObtenerCodigo @cod int output as
begin
    select @cod = Codigo
    from   Codigos holdlock
    update Codigos
    set    Codigo = Codigo + 1
end
go

-- Verificar cambio de signo a operación

create trigger CambioSigno on Operaciones for update as
if update(Ingreso)
    if exists (select * from Apuntes
        where Operacion in (select Codigo from deleted))
    begin
        raiserror('La operación tiene apuntes asociados', 16, 1)
        rollback transaction
    end
end
go

-- Mantenimiento del saldo

create trigger MantenerSaldo on Apuntes for insert as
begin
    declare @Libreta int, @Importe int, @Fecha datetime,
            @UltSaldo int, @UltApunte int, @UltFecha datetime,
            @TipoOp char

    select @Libreta = Libreta, @Importe = Importe,
           @Fecha = Fecha
    from   inserted
    select @TipoOp = Ingreso
    from   Operaciones
    where  Codigo = (select Operacion from inserted)
    if (@Importe < 0)
        select @Importe = - @Importe
    if (@TipoOp in ('N', 'n'))
        select @Importe = - @Importe

    select @UltSaldo = Saldo, @UltApunte = Apuntes,
           @UltFecha = Fecha
    from   Libretas holdlock
    where  Codigo = @Libreta

```

```

    if (@UltFecha is not null and @UltFecha > @Fecha)
    begin
        raiserror('Fecha de apunte incorrecta', 16, 1)
        rollback transaction
    end

    update Libretas
    set Saldo = Saldo + @Importe,
        Apuntes = Apuntes + 1,
        Fecha = @Fecha
    where Codigo = @Libreta
    update Apuntes
    set Numero = @UltApunte + 1,
        Importe = @Importe,
        Saldo = @UltSaldo + @Importe
    where Codigo = (select Codigo from inserted)
end
go

-- Modificación de apuntes

create procedure ModificarApunte @Cod int, @Imp int as
begin
    declare @Libreta int, @Numero int, @Importe int,
            @Operacion int, @TipoOp char

    -- Datos del apunte a corregir
    select @Libreta = Libreta, @Numero = Numero,
           @Importe = Importe, @Operacion = Operacion
    from Apuntes
    where Codigo = @Cod
    -- Corregir signo del nuevo importe
    if (@Imp < 0)
        select @Imp = - @Imp
    if (@TipoOp in ('N', 'n'))
        select @Imp = - @Imp
    -- Corregir el apunte actual
    update Apuntes
    set Importe = @Imp
    where Codigo = @Cod
    -- Propagar cambios en el saldo
    update Apuntes
    set Saldo = Saldo + @Imp - @Importe
    where Libreta = @Libreta and
           Numero >= @Numero
    -- Corregir resumen en libreta
    update Libretas
    set Saldo = Saldo + @Imp - @Importe
    where Codigo = @Libreta
end
go

```

Como es de suponer, existen diferencias importantes. El *trigger* para la actualización de los saldos se ejecuta después de realizada la grabación. Por este motivo tendremos que suministrar un valor por omisión para el número del apunte ... o renunciar a la unicidad de la combinación libreta/número. Muchos de estos *triggers*, además, asumen que @@RowCount, la variable global que nos dice cuántas filas han sido afecta-

das por la última instrucción SQL, vale 1. Observe también que hay que crear explícitamente los índices asociados a las relaciones de integridad referencial.

Ahora, en Oracle

El siguiente listado contiene la definición de los objetos de la base de datos, esta vez en el dialecto PL/SQL de Oracle. Estas son las instrucciones para la creación de tablas e índices:

```
-- Definición de las tablas -----

-- Operaciones
create table Operaciones (
    Codigo          int not null,
    Descripcion     varchar(35) not null,
    Ingreso         char(1),
    primary key     (Codigo),
    unique          (Descripcion),
    check           (Ingreso in ('S', 's', 'N', 'n'))
);

-- Libretas
create table Libretas (
    Codigo          int not null,
    Titular         varchar(35) not null,
    Descripcion     varchar(35) not null,
    Saldo           int default 0,
    Apuntes         int default 0,
    Fecha          date,
    primary key     (Codigo),
    unique          (Titular, Descripcion)
);

-- Apuntes
create table Apuntes (
    Codigo          int not null,
    Libreta         int not null,
    Numero          int not null,
    Fecha          date default sysdate not null,
    Operacion       int not null,
    Importe         int not null,
    Saldo           int,
    primary key     (Codigo),
    unique          (Libreta, Numero),
    foreign key     (Libreta) references Libretas(Codigo)
                    on delete cascade,
    foreign key     (Operacion) references Operaciones(Codigo)
);
create index FechaApunte on Apuntes(Fecha);
```

Los triggers y procedimientos almacenados se crean del siguiente modo:

```

-- Triggers y procedimientos almacenados -----
-- Códigos secuenciales

create sequence CodigosSeq increment by 1 start with 1;

-- Tabla de apuntes

create or replace procedure ObtenerCodigo(Codigo out int) as
begin
    select CodigosSeq.NextVal
    into   Codigo
    from   Dual;
end;
/

create or replace procedure ModificarApunte(unCod int,
unImp int) as
    Lib int;
    Num int;
    ImpAnt int;
begin
    select Libreta, Numero, Importe
    into   Lib, Num, ImpAnt
    from   Apuntes
    where  Codigo = unCod;
    update Libretas
    set    Saldo = Saldo + unImp - ImpAnt
    where  Codigo = Lib;
    update Apuntes
    set    Importe = unImp
    where  Codigo = unCod;
    update Apuntes
    set    Saldo = Saldo + unImp - ImpAnt
    where  Libreta = Lib
           and Numero >= Num;
end;
/

create or replace trigger NuevoApunte
before insert on Apuntes
for each row
declare
    TipoOp char;
    Ultima date;
begin
    -- Corregir importe de acuerdo al tipo de operación
    select Ingreso
    into   TipoOp
    from   Operaciones
    where  Codigo = :new.Operation;
    if (:new.Importe < 0) then
        :new.Importe := - :new.Importe;
    end if;
    if (TipoOp in ('N', 'n')) then
        :new.Importe := - :new.Importe;
    end if;
end if;

```

```

-- Actualizar la tabla de libretas
update Libretas
set     Saldo = Saldo + :new.Importe,
        Apuntes = Apuntes + 1
where   Codigo = :new.Libreta;
-- Recuperar el número de apunte y la fecha anterior
select Apuntes, Saldo, Fecha
into   :new.Numero, :new.Saldo, Ultima
from   Libretas
where  Codigo = :new.Libreta;
-- Verificar la fecha del nuevo apunte
if (Ultima is not null and Ultima > :new.Fecha) then
    raise_application_error(-20000,
        'Fecha de apunte incorrecta');
end if;
-- Modificar la fecha en la libreta
update Libretas
set     Fecha = Ultima
where   Codigo = :new.Libreta;
end;
/

create or replace trigger ModificarOperacion
before update on Operaciones
for each row
declare
Cant integer;
begin
-- Sólo se cambia el signo si no hay apuntes asociados
if (:new.Ingreso <> :old.Ingreso) then
    select count(*)
    into   Cant
    from   Apuntes
    where  Operacion = :old.Codigo;
    if (Cant > 0) then
        raise_application_error(-20002,
            'Esta operación tiene apuntes');
    end if;
end if;
end;
/

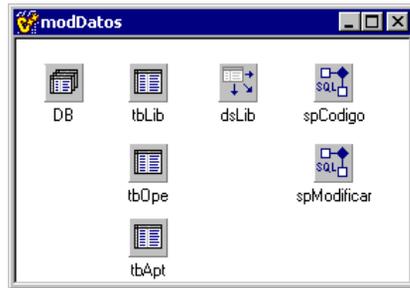
-- FIN -----

```

Aquí faltan los triggers que hacen posible el borrado de un apunte y la modificación de su fecha. El motivo: estos *triggers* deben hacer referencia a la tabla mutante, lo cual no es posible en Oracle. Para implementar dichas operaciones necesitaríamos procedimientos almacenados ... o *triggers* disparados después de la operación. Dejo este asunto a su cargo.

El módulo de datos

El desarrollo de la aplicación comenzará por la implementación del módulo de datos de la misma. El primer objeto será, por supuesto, un componente de tipo *TDatabase*, con el cual accederemos a la base de datos. La configuración del componente dependerá en lo fundamental del formato que se esté utilizando para los datos: InterBase, Oracle o MS SQL Server.



Una vez que tenemos la referencia a la base de datos traemos tres tablas al módulo: *tbOpe* (operaciones), *tbLib* (libretas) y *tbApt* (apuntes), y sus correspondientes fuentes de datos. Establecemos una relación *master/detail* desde la tabla de libretas hacia la tabla de apuntes:

Propiedad	Valor
<i>Name</i>	<i>tbApt</i>
<i>DatabaseName</i>	<i>APT</i>
<i>TableName</i>	<i>Apuntes</i>
<i>MasterSource</i>	<i>dsLib</i>
<i>MasterFields</i>	<i>Codigo</i>
<i>IndexFieldNames</i>	<i>Libreta;Numero</i>

Preste atención a la propiedad *IndexFieldNames*, pues se han concatenado los campos *Libreta* y *Numero*, para que la tabla de detalles quede ordenada efectivamente por el número de los apuntes.

He obviado la discusión acerca de qué tipo de componente utilizar para el acceso a datos: si tablas o consultas. Se trata de un ejemplo pequeño, por lo cual he recurrido directamente a las tablas.

Definimos también, en la tabla de apuntes, un campo de referencia a la tabla de operaciones. Como el modelo de datos de la aplicación es sencillo y pequeño, no utilizaremos una tabla separada para las referencias, que es lo que recomiendo en otros casos.

Es muy importante configurar correctamente la propiedad *Required* de cada campo. Por ejemplo, aunque la columna *Numero* de la tabla de apuntes se ha definido con la restricción **not null**, debemos dejar *Required* a *False*, pues el valor de la columna se asigna en un *trigger* en el servidor, y los registros de apuntes deben abandonar el cliente con un valor nulo en su número. Los campos que deben tener *Required* activo son los siguientes:

Tabla	Campos requeridos
<i>Libretas</i>	<i>Codigo, Titular, Descripcion</i>
<i>Operaciones</i>	<i>Codigo, Descripcion, Ingreso</i>
<i>Apuntes</i>	<i>Libreta, Fecha, Operacion, Importe</i>

Los restantes campos deben marcarse como *ReadOnly*, para que el usuario no pueda modificarlos directamente. Recuerde también cambiar las etiquetas de visualización y los formatos de visualización y edición de cada campo. Y si lo desea, puede aprovechar la propiedad *CustomConstraint* para exigir en el cliente que el importe de un nuevo apunte sea distinto de cero, y que las descripciones se llenen con cadenas no vacías.

Al haber utilizado dominios de InterBase para la creación de tablas, el BDE se confunde y no puede asignar correctamente la propiedad *Required* ... a no ser que primeramente importemos la definición de la base de datos en el Diccionario de Datos.

En la tabla de operaciones, por su parte, interceptaremos los eventos *OnGetText* y *OnSetText* del campo *Ingreso* para traducir el “sí” y el “no” en algo más legible, y para que el usuario pueda teclear la *I* de “ingreso” y la *R* de “reintegro” como valores aceptables:

```
void __fastcall TmodDatos::tbOpeINGRESOGetText(TField *Sender,
    AnsiString &Text, bool DisplayText)
{
    Text = "";
    if (! Sender->IsNull)
    {
        Text = UpperCase(Sender->AsString);
        if (DisplayText)
            Text = (Text == "S") ? "Ingreso" : "Reintegro";
    }
}

void __fastcall TmodDatos::tbOpeINGRESOSetText(TField *Sender,
    const AnsiString Text)
{
    AnsiString S = UpperCase(Text).SubString(1, 1);
    if (S == "")
        Sender->Clear();
    else if (S == "S" || S == "I")
        Sender->AsString = "S";
}
```

```

else if (S == "N" || S == "R")
    Sender->AsString = "N";
else
    // Dejar que falle la propia validación del campo
    Sender->AsString = S;
}

```

Traemos un par de componentes *TStoredProc* al módulo, *spCodigo* y *spModApt*, y hacemos que se refieran a los procedimientos almacenados *ObtenerCodigo* y *ModificarApunte*. El primer procedimiento almacenado será utilizado para asignar la clave primaria de un apunte antes de grabarlo, en el evento *BeforePost*:

```

void __fastcall TmodDatos::tbAptBeforePost(TDataSet *DataSet)
{
    if (tbAptCODIGO->IsNull)
    {
        spCodigo->ExecProc();
        tbAptCODIGO->Value =
            spCodigo->ParamByName("CODIGO")->AsInteger;
    }
}

```

Note que se pregunta si el valor del código es nulo, para no llamar innecesariamente al procedimiento almacenado cuando se trata de un reintento de grabación o durante una simple modificación.

Los últimos tres eventos que manejaremos están relacionados con la inicialización del campo *Fecha* de los apuntes durante la inserción. Es muy conveniente que, cuando estemos tecleando apuntes, se mantenga para la fecha del nuevo apunte la fecha del último apunte insertado. Esto lo conseguimos declarando una variable *UltimaFecha* en el módulo de datos:

```

class TmodDatos : public TDataModule
{
    // ...
private:
    TDateTime UltimaFecha;
    // ...
};

```

Estos son, finalmente, los eventos que utilizan esta variable:

```

void __fastcall TmodData::tbAptNewRecord(TDataSet *DataSet)
{
    tbAptFECHA->Value = UltimaFecha;
}

void __fastcall TmodData::tbAptAfterPost(TDataSet *DataSet)
{
    UltimaFecha = tbAptFECHA->Value;
}

```

```

void __fastcall TmodDatos::dsLibDataChange(TObject *Sender,
    TField *Field)
{
    UltimaFecha = tbLibFECHA->IsNull ? Date() : tbLibFECHA->Value;
}

__fastcall TmodData::TmodData(TComponent *Owner)
    : TDataModule(Owner)
{
    UltimaFecha = tbLibFECHA->IsNull ? Date() : tbLibFECHA->Value;
}

```

Transacciones explícitas

InterBase no tiene instrucciones para colocar explícitamente bloqueos durante una transacción ... pero tampoco le hace falta. Para garantizar que las operaciones complejas de alta de apuntes y de modificación de importe sean coherentes, debemos ejecutarlas durante una transacción con el nivel de aislamiento de lecturas repetibles. Pero las transacciones implícitas de InterBase que implementa el BDE transcurren con el nivel *tiReadCommitted*, a no ser que activemos el valor 512 en el parámetro *DRIVER_FLAGS* del controlador. Y en ese último caso se nos complica la semántica de los mantenimientos tradicionales de tablas, pues hace falta iniciar una nueva transacción cada vez que necesitemos “ver” los cambios efectuados desde otros puestos.

Por lo tanto, modificaremos el nivel de aislamiento del componente *BD*, de tipo *TDatabase*, asignando el valor *tiRepeatableRead* a su propiedad *TransIsolation*. Después añadiremos a la declaración de la clase del módulo de datos un par de métodos públicos con las implementaciones que muestro a continuación:

```

void __fastcall TmodDatos::GrabarApunte(void)
{
    DB->StartTransaction();
    try {
        tbApt->Post();
        DB->Commit();
    }
    catch(Exception& E) {
        DB->Rollback();
        throw;
    }
}

void __fastcall TmodDatos::ModificarApunte(int Importe)
{
    DB->StartTransaction();
    try
    {
        spModificar->ParamByName("COD")->AsInteger =
            tbAptCODIGO->Value;
        spModificar->ParamByName("IMP")->AsInteger = Importe;
        spModificar->ExecProc();
    }
}

```

```

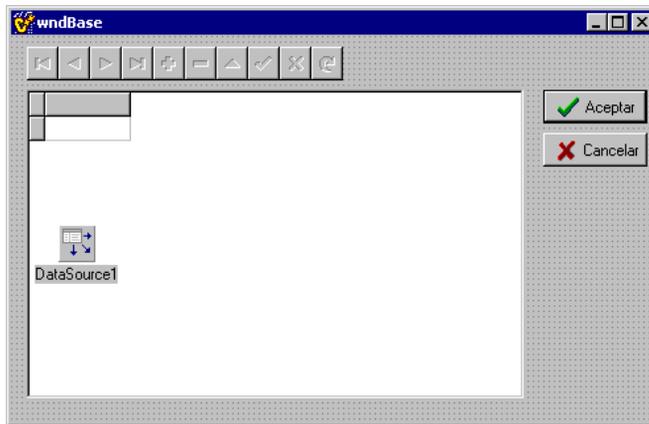
        DB->Commit();
    }
    catch(Exception& E) {
        DB->Rollback();
        throw;
    }
}

```

Gestión de libretas y operaciones

Pasamos ahora al desarrollo visual de la aplicación. La ventana principal será una ventana SDI, que mostrará los apuntes correspondientes a una libreta seleccionada; no olvide que la tabla de apuntes está en relación *master/detail* con la de libretas. Antes de mostrar la ventana principal de la aplicación necesitaremos un cuadro de diálogo que nos permita gestionar libretas (crear, modificar y eliminar) y seleccionar una de ellas antes de seguir trabajando. Por lo tanto, comenzaremos con la creación de estos formularios.

Como el mantenimiento de libretas y operaciones se realizará sobre ventanas muy parecidas entre sí, crearemos una ventana “base”, a partir de la cual se crearán las otras dos por medio de la herencia visual. El modelo de ventana base que utilizaremos puede ser parecido al de la imagen que muestro a continuación:



Los formularios de mantenimiento se utilizarán también para la selección de registros. Nos ocuparemos de guardar la posición inicial del cursor de la tabla asociada cuando se cree dinámicamente el formulario; más adelante, si el usuario cancela la ejecución, debemos regresar a dicha posición:

```

void __fastcall TwndBase::FormCreate(TObject *Sender)
{
    Marca = DataSource1->DataSet->Bookmark;
}

```

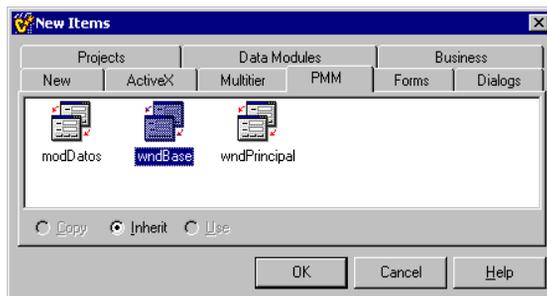
Cuando se cierra el formulario debemos hacer dos cosas: grabar o cancelar cualquier cambio pendiente y, si se ha cancelado la ejecución, intentar el regreso a la posición inicial del cursor.

```
void __fastcall TwndBase::FormCloseQuery(TObject *Sender,
    bool &CanClose)
{
    if (ModalResult == mrOk)
        // Grabar cualquier posible cambio
        DataSource1->DataSet->CheckBrowseMode();
    else
    {
        DataSource1->DataSet->Cancel();
        // Intentar regresar a la posición original
        try { DataSource1->DataSet->Bookmark = Marca; }
        catch(...) {}
    }
}
```

Y para simplificar la creación y destrucción de estos formularios, interceptamos también el evento *OnClose*:

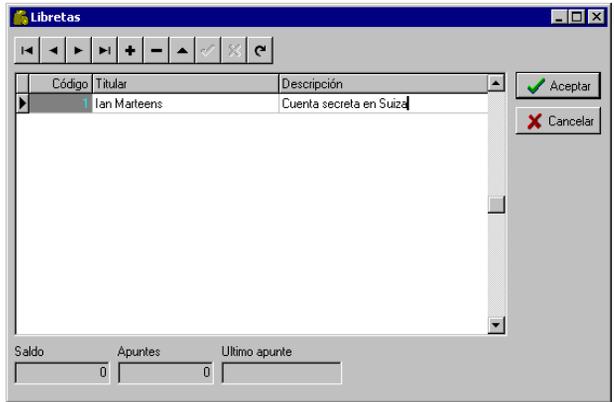
```
void __fastcall TwndBase::FormClose(TObject *Sender,
    TCloseAction &Action)
{
    Action = caFree;
}
```

A continuación derivaremos la ventana de gestión de libretas a partir del formulario *wndBase* por medio de la herencia visual:



Bautizaremos a la nueva ventana como *dlgLibretas*, y guardaremos la nueva unidad con el nombre *Libretas*. Para personalizar el formulario asociamos la tabla *tblLib* del módulo de datos al componente *DataSource1*. Después, con un doble clic sobre la rejilla, configuramos las columnas de forma tal que solamente aparezcan en la misma los campos editables: *Codigo*, *Titular* y *Descripcion*. Los campos redundantes se mostrarán en componentes *DBEdit* situados debajo de la rejilla: *Apuntes*, *Saldo* y *Fecha*. Los tres editores deben tener *True* en su propiedad *ReadOnly*. El motivo de este trato dis-

criminatorio es, en parte, que no me gusta colocar columnas dentro de una rejilla si esto me fuerza a utilizar una barra de desplazamiento horizontal.

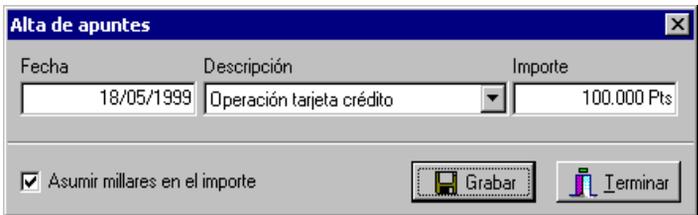


La creación de la ventana de mantenimiento de operaciones es igual de sencilla, con la salvedad de que podemos dejar todos los campos de la tabla para que sean editados en la rejilla de datos.

Entrada de apuntes

Si queremos que alguien utilice nuestra maravillosa aplicación, tendremos que esmerarnos con el cuadro de entrada de apuntes; lo sé por experiencia personal. Debemos hacer que la introducción de datos sea lo más cómoda posible, teniendo en cuenta que es la parte de la aplicación que más se utilizará.

Creamos un nuevo formulario, al cual llamaremos *wndApuntes*, y este será su aspecto:



Durante la ejecución del evento *OnCreate*, haremos que la tabla de apuntes entre en el estado de inserción:

```
void __fastcall TwndApuntes::FormCreate(TObject *Sender)
{
    modDatos->tbApt->Append();
}
```

La grabación y cancelación de cambios se realiza en respuesta a los eventos *OnClick* del botón de grabación y *OnClose*, y es una variación del esquema que hemos estado utilizando a lo largo del libro:

```
void __fastcall TwndApuntes::bnGrabarClick(TObject *Sender)
{
    modDatos->GrabarApunte();
    modDatos->tbApt->Append();
    SelectFirst();
}

void __fastcall TwndApuntes::FormClose(TObject *Sender,
    TCloseAction &Action)
{
    modDatos->tbApt->Cancel();
    modDatos->tbLib->Refresh();
    Action = caFree;
}
```

Para facilitar la entrada de datos numéricos, transformaremos la acción de la tecla INTRO, de modo que seleccione el próximo control del diálogo. Primero, asignamos *True* a la propiedad *KeyPreview*, para después interceptar el evento *OnKeyPress* del formulario:

```
void __fastcall TwndApuntes::FormKeyPress(TObject *Sender,
    char &Key)
{
    if (Key == 13 &&
        ! ActiveControl->InheritsFrom(__classid(TButton)))
    {
        TDBLookupComboBox *cb =
            dynamic_cast<TDBLookupComboBox*>(ActiveControl);
        if (cb && cb->ListVisible)
            cb->CloseUp(True);
        else
            SelectNext(ActiveControl, True, True);
        Key = 0;
    }
}
```

También ayudaremos al usuario cuando se encuentre sobre el cuadro de edición de fechas, *DBEdit1* en nuestro formulario. Queremos que las teclas + y - sirvan para pasar al día siguiente y al día anterior, respectivamente. La clave es manejar el evento *OnKeyPress*, esta vez del control *DBEdit1*:

```
void __fastcall TwndApuntes::DBEdit1KeyPress(TObject *Sender,
    char &Key)
{
    TDateTimeField *f = modDatos->tbAptFECHA;
    switch (Key) {
        case '+': f->Value = f->Value + 1; break;
        case '-': f->Value = f->Value - 1; break;
        default: return;
    }
}
```

```

    Key = 0;
}

```

Como habrá observado el lector, el formulario tiene una casilla de verificación, a la cual hemos bautizado *bxMillares*; cuando esté activa, los importes menores que 1000 pesetas que se tecleen en el cuadro de edición *DBEdit2*, se multiplicarán automáticamente por 1000:

```

void __fastcall TwndApuntes::DBEdit2Exit(TObject *Sender)
{
    TIntegerField *f = modDatos->tbAptIMPORTE;
    if (! f->IsNull && bxMillares->Checked && abs(f->Value) < 1000)
        f->Value *= 1000;
}

```

El lector puede añadir sus propias mejoras al formulario. Un calendario emergente, por ejemplo, sería de mucha utilidad para la edición de fechas.

La ventana principal

Ya podemos plantearnos la creación de la ventana principal. El aspecto final de la misma se muestra en la siguiente imagen:



El primer paso es mostrar la ventana de gestión de libretas antes de mostrar la ventana principal en la pantalla. Para esto aprovecharemos el evento *OnShow* del formulario:

```

void __fastcall TwndPrincipal::FormShow(TObject *Sender)
{
    if ((new TwndLibretas(0))->ShowModal() != mrOk ||
        modDatos->tbLib->IsEmpty())
        Application->Terminate();
}

```

Para el diseño del menú y las barras de herramientas he utilizado las acciones introducidas en C++ Builder 4, pero el lector puede utilizar sin problema su método favorito de creación de interfaces visuales. El menú de la ventana, creado de una forma u otra, será el siguiente:

Fichero	Ver	Ayuda
Libretas...	Saldo...	Acerca de...
Operaciones...	Análisis...	
Altas... Ctrl+A		
Bajas... Ctrl+Supr		
Modificar importe... Ctrl+M		
Configurar impresora...		
Imprimir Ctrl+P		
Salir		

La visualización de los formularios *wndLibretas* y *wndOperaciones* es extremadamente simple:

```
void __fastcall TwndPrincipal::acLibretasExecute(TObject *Sender)
{
    (new TwndLibretas(0))->ShowModal();
}

void __fastcall TwndPrincipal::acOperacionesExecute(TObject *Sender)
{
    (new TwndOperaciones(0))->ShowModal();
}
```

Recuerde que ambos formularios se destruyen automáticamente al ser cerrados, y que ambos devuelven la fila activa de la tabla asociada al registro inicialmente seleccionado, si es que su ejecución es cancelada. El lanzamiento del diálogo de entrada de apuntes es parecido:

```
void __fastcall TwndPrincipal::acAltasExecute(TObject *Sender)
{
    (new TwndApuntes(0))->ShowModal();
}
```

El siguiente método, que se ejecuta en respuesta al evento *OnDrawColumnCell* de la rejilla, se ocupa de mostrar en verde o en rojo los valores del saldo y del importe de cada apunte, según el valor de la columna sea positivo o negativo:

```
void __fastcall TwndPrincipal::DBGrid1DrawColumnCell(
    TObject *Sender, const TRect &Rect, int DataCol, TColumn *Column,
    TGridDrawState State)
{
    TFont* f = DBGrid1->Canvas->Font;
    if (! State.Contains(gdFocused))
        if (CompareText(Column->FieldName, "SALDO") == 0)
            f->Color = modDatos->tbAptSALDO->Value > 0 ?
```

```

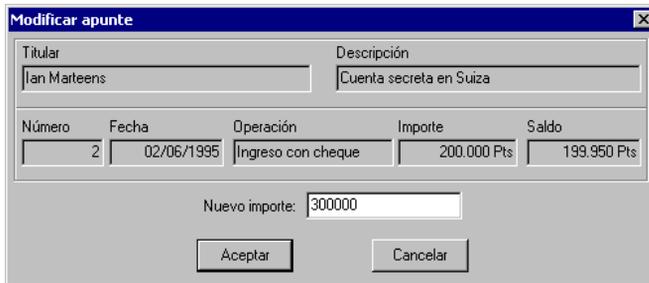
        clGreen : clRed;
    else if (CompareText(Column->FieldName, "IMPORTE") == 0)
        f->Color = modDatos->tbAptIMPORTE->Value > 0 ?
            clGreen : clRed;
    DBGrid1->DefaultDrawColumnCell(Rect, DataCol, Column, State);
}

```

Corrigiendo el importe de un apunte

Sin embargo, y a pesar de todas las facilidades que podamos incluir para la entrada de apuntes, es normal que el usuario pueda equivocarse tecleando un apunte, sobre todo en lo que respecta al importe. Yo, por ejemplo, tiendo a pagar menos y a cobrar más. En circunstancias reales, cuando el banco se equivoca se introducen apuntes compensatorios. Esta solución no es la mejor para los objetivos de la aplicación, pues este tipo de apuntes complica la obtención de estadísticas. Por fortuna, esta situación la habíamos previsto, cuando creamos el procedimiento almacenado *ModificarApunte* en la base de datos.

Necesitamos un nuevo cuadro de diálogo, al cual le daremos el nombre de *wndModificar*. La siguiente imagen muestra el aspecto visual del mismo:



Los cuadros de edición que aparecen desactivados en la parte superior del diálogo, son controles *DBEdit* enlazados a los campos de la tabla *tbApt*, y sirven de referencia para la modificación del apunte. En la creación del formulario, se inicializa el texto del control *Edit1* (el nuevo importe) con el valor actual del importe del apunte a modificar:

```

void __fastcall TwndModificar::FormCreate(TObject *Sender)
{
    Edit1->Text = modDatos->tbAptIMPORTE->AsString;
}

```

Cuando se vaya a cerrar el cuadro de diálogo, se ejecuta el procedimiento almacenado *spModApt*, que hemos incluido en el módulo de datos:

```

void __fastcall TwndModificar::FormCloseQuery(TObject *Sender,
    bool &CanClose)
{
    if (ModalResult != mrOk) return;
    // Ejecutar el procedimiento almacenado
    modDatos->ModificarApunte(StrToInt(Edit1->Text));
}

```

Este cuadro de diálogo se lanza desde la ventana principal, en respuesta al comando *Modificar importe* del menú *Fichero*:

```

void __fastcall TwndMain::ModificarImporte1Click(TObject *Sender)
{
    std::auto_ptr<TdlgModApt> dlg(new TwndModificar(0));
    dlg->ShowModal();
}

```


Ejemplos: un servidor de Internet

PARA PONER EN ORDEN TODA LA INFORMACIÓN QUE aprendimos en el capítulo 38, desarrollaremos un pequeño ejemplo de extensión de servidor Web. Con este servidor podremos buscar en una base de datos de productos de software mediante una lista de palabras claves. La lista de palabras, para simplificar, será creada explícitamente por el administrador de la base de datos cuando inserte productos en la base de datos.

Búsqueda de productos

A continuación incluyo un fichero *script* para InterBase con la definición de la base de datos y las tablas que utilizaremos:

```
create database "C:\La Cara Oculta de Cppb\Datos\Productos.GDB"
user "SYSDBA" password "masterkey" page_size 1024;
set autoddl on;

/** Definición de las tablas *****/

create table Productos (
   Codigo          int not null,
   Descripcion     varchar(45) not null,
   Categoria       varchar(30) not null,
   UCat            varchar(30) not null,
   Comentario      blob sub_type 1,

    primary key    (Codigo),
    unique         (Descripcion)
);
create index CategoriaProducto on Productos(UCat);

create table Palabras (
   Codigo          int not null,
   Palabra         varchar(30) not null,

    primary key    (Codigo),
    unique         (Palabra)
);
```

```

create table Claves (
    Producto      int not null,
    Palabra       int not null,

    primary key   (Producto, Palabra),
    foreign key   (Producto) references Productos(Codigo)
                   on update cascade on delete cascade,
    foreign key   (Palabra) references Palabras(Codigo)
                   on update cascade on delete cascade
);

/** Triggers y procedimientos almacenados *****/

set term ^;

create generator GenProductos^

create procedureCodigoProducto returns (Codigo int) as
begin
    Codigo = gen_id(GenProductos, 1);
end^

create trigger NuevoProducto for Productos
active before insert as
begin
    new.UCat = upper(new.Categoria);
end^

create procedure BorrarPalabras(Producto int) as
begin
    delete from Claves
    where Producto = :Producto;
end^

/* Palabras y asociaciones */

create generator GenPalabras^

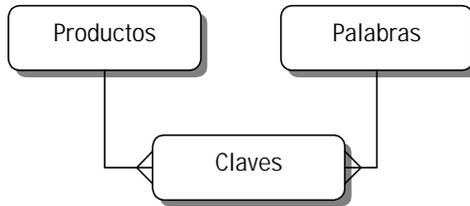
create procedure Asociar(Producto int, Palabra varchar(30)) as
declare variable CodPal int;
begin
    Palabra = upper(Palabra);
    select Codigo
    from Palabras
    where Palabra = :Palabra
    into :CodPal;
    if (CodPal is null) then
    begin
        CodPal = gen_id(GenPalabras, 1);
        insert into Palabras
        values (:CodPal, :Palabra);
    end
    insert into Claves(Producto, Palabra)
    values (:Producto, :CodPal);
end^

set term ;^

```

La tabla fundamental es la de productos, que asocia a cada producto un código numérico, una descripción, comentarios y una categoría. La categoría es una cadena de la cual se guardan dos versiones: una en mayúsculas (*UCat*) y la otra utilizando los mismos caracteres introducidos por el usuario. Este es un truco aplicable a InterBase, pero también a Oracle y otros sistemas que no tienen índices insensibles a mayúsculas y minúsculas, y sirve para acelerar las búsquedas: observe que el índice se ha creado sobre el campo *UCat*.

Paralelamente, existe una tabla con todas las palabras claves que están asociadas a los productos. Esta tabla se encarga de asociar un código numérico a cada una de las palabras. Finalmente, la tabla *Claves* almacena pares formados por un código de producto y un código de palabra. En general, cada producto puede estar asociado a varias palabras, y cada palabra puede estar vinculada con varios productos.

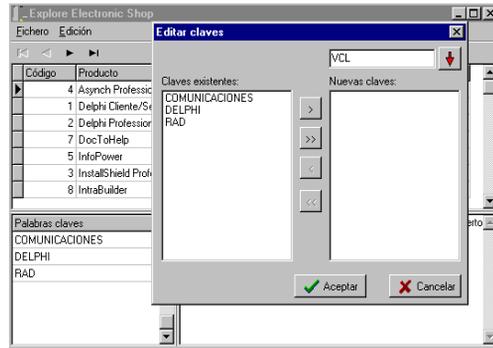


No voy a incluir la búsqueda por categorías, pues es una extensión sencilla al programa que vamos a desarrollar.

El motor de búsquedas

El paso siguiente consiste en crear una pequeña aplicación para poblar la base de datos de productos. La aplicación debe permitir introducir productos y sus descripciones, y asociarle posteriormente palabras claves para la búsqueda. Como es una aplicación muy sencilla, he decidido no incluir todo el código de la misma en el texto del libro, pero el usuario puede encontrar las fuentes del programa, en el CD-ROM adjunto.

Como parte importante de la aplicación, incluiremos la posibilidad de que el usuario compruebe las asociaciones realizadas entre productos y palabras claves mediante una ventana de búsqueda. La parte fundamental de este mecanismo de búsqueda consiste en una función, de nombre *GenerateQuery*, que partiendo de una cadena con la lista de palabras a buscar separadas por espacios genera una instrucción **select**; esta consulta se copia dentro de un componente *Query*, que al abrirse muestra sus resultados en una rejilla de datos.



Esta función será utilizada tanto por la aplicación Web como por la utilidad que sirve para poblar la base de datos. Por lo tanto, se ha aislado, junto con la función auxiliar *Tokenize*, dentro de una unidad denominada *SearchEngine*, que deberemos incluir en el proyecto Web más adelante. Este es el código de ambas funciones:

```

void __fastcall Tokenize(AnsiString AText, TStrings *AList)
{
    AText = AnsiUpperCase(AText.Trim());
    while (AText != "")
    {
        int i;
        AnsiString S;
        if (AText[1] == '\\')
        {
            AText.Delete(1, 1);
            i = AText.Pos("\\");
        }
        else
            i = AText.Pos(" ");
        if (i == 0)
        {
            S = AText;
            AText = "";
        }
        else
        {
            S = AText.SubString(1, i - 1);
            AText.Delete(1, i);
            AText = AText.TrimLeft();
        }
        if (S != "") AList->Add(S);
    }
}

void __fastcall GenerateQuery(const AnsiString AText,
    TStrings *AQuery)
{
    std::auto_ptr<TStrings> AList(new TStringList);
    Tokenize(AText, AList.get());
    AQuery->Clear();
    AQuery->Add("SELECT * FROM PRODUCTOS");
    AnsiString Prefix = "WHERE ";
}

```

```

for (int i = 0; i < AList->Count; i++)
{
    AQuery->Add(Prefix + "CODIGO IN ");
    AQuery->Add("      (SELECT PRODUCTO");
    AQuery->Add("      FROM CLAVES, PALABRAS");
    AQuery->Add("      WHERE CLAVES.PALABRA=PALABRAS.CODIGO");
    AQuery->Add("      AND PALABRAS.PALABRA = " +
        QuotedStr(AnsiUpperCase(AList->Strings[i].Trim())) + " ");
    Prefix = " AND ";
}
}

```



Para comprender cómo funciona este procedimiento, supongamos que el usuario teclea el siguiente texto en el control de palabras a buscar:

```
"bases de datos" rad vcl
```

El primer paso que realiza *GenerateQuery* es descomponer la cadena anterior en las palabras o expresiones a buscar. El resultado intermedio es la siguiente lista de cadenas:

```

bases de datos
rad
vcl

```

A partir de la lista anterior, se va creando una instrucción **select** en la que se añade una subconsulta sobre las tablas de claves y palabras por cada palabra tecleada por el usuario. La consulta generada es la siguiente:

```

select *
from Productos
where Codigo in
    (select Producto
     from Claves, Palabras
     where Claves.Palabra = Palabras.Codigo
        and Palabras.Palabra = 'BASES DE DATOS')
and Codigo in
    (select Producto
     from Claves, Palabras
     where Claves.Palabra = Palabras.Codigo
        and Palabras.Palabra = 'RAD')

```

```

and Codigo in
(select Producto
 from Claves, Palabras
 where Claves.Palabra = Palabras.Codigo
 and Palabras.Palabra = 'VCL')

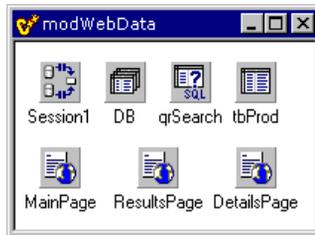
```

Seguramente que el lector estará pensando en mejoras a este generador. Por ejemplo, incluir la búsqueda por categorías, permitir búsquedas parciales por prefijos de palabras o incluir otros operadores lógicos además de la conjunción.

Creando la extensión Web

Comenzaremos en este momento la creación de la extensión de servidor Web. Ejecute el comando *File|New* para desplegar el Depósito de Objetos, y seleccione el icono *Web Server Application*. Luego elija una aplicación ISAPI/NSAPI ó CGI, según prefiera. Si está utilizando C++ Builder 3, recuerde modificar el texto del proyecto, si elige ISAPI/NSAPI, como hemos explicado en el capítulo 38, para indicar que esta aplicación lanzará múltiples subprocesos.

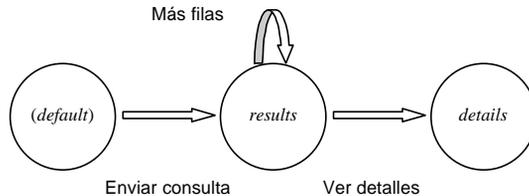
Cambie el nombre del módulo de datos a *modWebData*, y coloque los objetos que aparecen a continuación:



El primer objeto a situar es *Session1*, para que cada hilo concurrente del servidor corresponda a un usuario diferente y no tengamos problemas de sincronización. Debemos asignar *True* a su propiedad *Auto.SessionName*, para evitar conflictos con las sesiones concurrentes; recuerde que cada cliente conectado va a crear una nueva instancia del módulo de datos. Después traemos la base de datos *BD*, la enganchamos al objeto de sesión mediante su propiedad *SessionName*, y configuramos sus parámetros de modo que podamos conectarnos a la base de datos, en la ubicación en que se encuentre en el servidor de Internet. Finalmente, he conectado una consulta, *qrSearch*, y una tabla, *tbProd*, a la base de datos, ambas inactivas en tiempo de diseño. La consulta tiene su instrucción SQL sin asignar, pues se generará mediante la función *GenerateQuery* que hemos definido antes. Por su parte, *tbProd* debe conectarse a la tabla de productos de la base de datos, y se utilizará en la generación de la página de detalles de un producto.

El flujo de la aplicación será el siguiente:

- El usuario se conecta a la página principal (acción por omisión). En esta página se le pedirá que teclee la cadena de búsqueda. Cuando pulse el botón de envío, se intentará activar la acción *results*, para ver los resultados de la búsqueda.
- La página generada con la acción *results* mostrará la información de cabecera de los productos dentro de una tabla HTML. Limitaremos el número de filas por página, de modo que en determinados casos debe aparecer un botón al finalizar la página para recuperar más filas.
- En la página de resultados, cada fila de productos tiene un enlace en la columna del nombre del producto para que amplíemos la información sobre el producto seleccionado. Esta información se mostrará en una tercera página, correspondiente a la acción *details*.



En concordancia con el flujo de páginas en la aplicación definimos tres acciones en el módulo, como puede verse en la siguiente imagen:

Name	PathInfo	Enabled	Default
Main		True	*
Results	/results	True	
Details	/details	True	

Ahora definiremos constantes y funciones auxiliares. Cuando generemos el texto HTML, tendremos que hacer referencias a otras páginas situadas dentro del servidor. Aunque podemos utilizar una URL relativa, prefiero siempre una URL completa. Pero esa dirección depende de si estamos probando la extensión en nuestro servidor local o en el servidor real de Internet. Por eso, he añadido una declaración de cadena de recursos, para indicar en qué dominio se encuentra nuestro programa:

```

const
  AnsiString SDominio = "http://ian.marteens.com";
  
```

La siguiente función utiliza la cadena definida para generar una URL:

```

AnsiString __fastcall TmodWebData::GenURL(const AnsiString APath)
{
    return SDominio + "/scripts/ShopSrvr.dll/" + APath;
}

```

También he apartado en métodos auxiliares la generación de cadenas HTML que utilizaremos con frecuencia. Por ejemplo, el encabezamiento de formularios:

```

AnsiString __fastcall TmodWebData::GenFormHeader(
    const AnsiString APath)
{
    return "<FORM ACTION=\"" + GenURL(APath) + "\" METHOD=GET>";
}

```

Los tres objetos de clase *TPageProducer* serán analizados más adelante.

Generando la tabla de resultados

Seguimos definiendo métodos auxiliares en el módulo. La función *GenResults*, que mostraremos a continuación, genera el fragmento de código HTML con la tabla de productos encontrados a partir de la consulta del usuario. Esta función tiene un requisito importante: la consulta *qrSearch* debe estar posicionada en el primer registro que queremos que aparezca en la tabla. El motivo es fácil de entender: una consulta puede generar muchos productos coincidentes, por lo cual se tratará de limitar la cantidad de productos por página, mediante la constante *MAXLINKS*, definida globalmente en el módulo:

```
const int MAXLINKS = 5;
```

Uno de los parámetros que pasaremos en la URL será el último código de producto listado. Evidentemente, cuando el usuario realiza su consulta todavía no ha recibido ningún listado, por lo que este parámetro tendrá un valor por omisión: *-1*. Si un listado de productos sobrepasa el máximo de filas, colocaremos un botón de enlace al final del mismo, que ejecutará nuevamente la consulta, pero pasándole un parámetro *LastCode* que indique cuál es el último código de producto incluido en la tabla. El siguiente listado muestra los detalles de la composición del documento HTML:

```

AnsiString __fastcall TmodWebData::GenResults()
{
    int LastCode;

    if (qrSearch->Eof)
        return "<HTML><BODY><P>"
            "No se encontró ningún producto.</P></BODY></HTML>";
    AnsiString Rslt = "<Table Width=100% Border=1>"
        "<TR><TH BgColor=\"Blue\">CODIGO</TH>"
        "<TH BgColor=\"Blue\">PRODUCTO</TH>"
        "<TH BgColor=\"Blue\">CATEGORIA</TH></TR>";
}

```

```

for (int i = 0; i < MAXLINKS; ++i)
{
    AppendStr(Rslt,
        "<TR><TD>" + qrSearch->FieldByName("CODIGO")->AsString +
        "</TD><TD><A HREF=\"\" + GenURL("details") +
        "?Codigo=" + qrSearch->FieldByName("CODIGO")->AsString +
        "\">" + qrSearch->FieldValues["DESCRIPCION"] +
        "</A></TD><TD>" +
        qrSearch->FieldValues["CATEGORIA"] + "</TD></TR>");
    qrSearch->Next();
    if (qrSearch->Eof) {
        LastCode = -1;
        break;
    }
    LastCode = qrSearch->FieldValues["CODIGO"];
}
AppendStr(Rslt, "</Table>");
if (LastCode != -1) {
    AppendStr(Rslt, GenFormHeader("results") +
        "<INPUT TYPE=\"HIDDEN\" NAME=\"LASTCODE\" \" +
        \"VALUE=\"\" + IntToStr(LastCode) + \"\>\" +
        "<INPUT TYPE=\"HIDDEN\" NAME=\"Keywords\" VALUE=\"\" +
        Request->QueryFields->Values["Keywords"] + "\">\" +
        "<INPUT TYPE=\"SUBMIT\" \" +
        \"VALUE=\"Más productos\"></FORM>");
}
return Rslt;
}

```

Documentos HTML y sustitución de etiquetas

Implementaremos ahora la sustitución de etiquetas en los generadores de contenidos, asociando respuestas a sus eventos *OnHTMLTag*. Comenzaremos por el componente *MainPage*, que contiene el texto de la página principal en su propiedad *HTMLDoc*:

```

<HTML><BODY>
<H2>¡Bienvenido al Centro de Información de Productos!</H2>
<HR>
<#FORMHEADER>
<P>Teclee hasta 10 palabras para realizar la búsqueda:
<INPUT NAME="Keywords"></P>
<INPUT TYPE="HIDDEN" NAME="LASTCODE" VALUE="-1">
<INPUT TYPE="SUBMIT" VALUE="Buscar">
</FORM>
</BODY></HTML>

```

La página tiene una sola etiqueta transparente, `<#FORMHEADER>`, cuya sustitución es simple:

```

void __fastcall TmodWebData::MainPageHTMLTag(
    TObject *Sender, TTag Tag, const AnsiString TagString,
    TStringList *TagParams, AnsiString &ReplaceText)
{
    if (TagString == "FORMHEADER")
        ReplaceText = GenFormHeader("results");
}

```

Los resultados de la búsqueda se mostrarán en la siguiente página, contenida en el componente *ResultsPage*:

```

<HTML>
<BODY>
<H2>Resultados de la búsqueda</H2>
<HR>
<#RESULTS>
</BODY>
</HTML>

```

Bien, la sustitución de la etiqueta <#RESULTS> es complicada, pero ya hemos recorrido más de la mitad del camino en la sección anterior:

```

void __fastcall TmodWebData::ResultsPageHTMLTag(
    TObject *Sender, TTag Tag, const AnsiString TagString,
    TStringList *TagParams, AnsiString &ReplaceText)
{
    if (TagString == "RESULTS")
        ReplaceText = GenResults();
}

```

Finalmente, así se verá la información detallada acerca de un producto, mediante el componente *DetailsPage*:

```

<HTML><BODY>
<P><H3>Nombre del producto: </H3><#PRODUCTO></P>
<P><H4>Categoría: </H4><#CATEGORIA></P>
<H4>Comentarios:</H4>
<#COMENTARIOS>
</BODY></HTML>

```

Y este será el método que sustituirá las etiquetas correspondientes:

```

void __fastcall TmodWebData::DetailsPageHTMLTag(
    TObject *Sender, TTag Tag, const AnsiString TagString,
    TStringList *TagParams, AnsiString &ReplaceText)
{
    if (TagString == "PRODUCTO")
        ReplaceText = tbProdDescripcion->Value;
    else if (TagString == "CATEGORIA")
        ReplaceText = tbProdCategoría->Value;
    else if (TagString == "COMENTARIOS")
    {
        std::auto_ptr<TStrings> AList(new TStringList);
        AList->Assign(tbProdComentario);
        ReplaceText = "<BLOCKQUOTE>";
    }
}

```

```

    for (int i = 0; i < AList->Count; i++)
        AppendStr(ReplaceText, "<P>" + AList->Strings[i] + "</P>");
    AppendStr(ReplaceText, "</BLOCKQUOTE>");
}
}

```

He utilizado un componente *TPageProducer* en la generación de la página de detalles por compatibilidad con C++ Builder 3. La versión 4 introduce el componente *TDataSetPageProducer*, similar a *TPageProducer*, pero que se asocia a un conjunto de datos, y puede sustituir automáticamente las etiquetas que correspondan a nombres de columnas con el valor de las mismas en la fila actual del conjunto de datos.

Respondiendo a las acciones

Por último, aquí están los métodos de respuesta a las acciones, que sirven para integrar todos los métodos desarrollados anteriormente:

```

void __fastcall TmodWebData__modWebDataStartAction(
    TObject *Sender, TWebRequest *Request, TWebResponse *Response,
    bool &Handled)
{
    Response->Content = MainPage->Content();
}

void __fastcall TmodWebData::modWebDataResultsAction(
    TObject *Sender, TWebRequest *Request, TWebResponse *Response,
    bool &Handled)
{
    Request->ExtractContentFields(Request->QueryFields);
    int LastCode = StrToInt(
        Request->QueryFields->Values["LASTCODE"]);
    GenerateQuery(Request->QueryFields->Values["Keywords"],
        qrSearch->SQL);
    qrSearch->Open();
    try {
        if (LastCode > 0)
            qrSearch->Locate("CODIGO", LastCode, TLocateOptions());
        Response->Content = ResultsPage->Content();
    }
    __finally {
        qrSearch->Close();
    }
}

void __fastcall TmodWebData::modWebDataDetailsAction(
    TObject *Sender, TWebRequest *Request, TWebResponse *Response,
    bool &Handled)
{
    Request->ExtractContentFields(Request->QueryFields);
    tbProd->Open();
}

```

```
    try {
        tbProd->Locate("CODIGO",
            Request->QueryFields->Values["Codigo"], TLocateOptions());
        Response->Content = DetailsPage->Content();
    }
    finally {
        tbProd->Close();
    }
}
```

Y esto es todo, amigos.

APENDICE: Excepciones

SI CON LA PROGRAMACIÓN ORIENTADA A OBJETOS aprendimos a organizar nuestros datos y algoritmos, si con los Sistemas Controlados por Eventos nos liberamos de la programación “posesiva”, con las Excepciones daremos el último paso necesario: aprenderemos a ver a un programa como un proceso que se desarrolla en *dos* dimensiones. Y nos liberaremos de los últimos vestigios de la neurosis “yo lo controlo todo en mi programa”, un paso importante hacia la Programación Zen.

Sistemas de control de errores

Errar es de humanos, pero para meter de verdad la pata hace falta un ordenador. O, como dice la famosa ley de Murphy: todo lo que puede fallar, falla. Acabamos de comprar una biblioteca de funciones para trabajar, digamos, con la nueva NDAPI (*Nuclear Devices Application Programming Interface*). Por motivos de seguridad, en adelante utilizaremos nombres falsos y cortos para las funciones de la biblioteca; también omitiremos los parámetros. Supongamos que cierto proceso que está usted programando requiere una secuencia lineal de llamadas a rutinas:

```
A(); B(); C(); D(); E();
```

Ahora bien, cada una de estas rutinas puede fallar por ciertos y determinados motivos. Para controlar los errores, cada rutina es una función que devuelve un código entero representando el estado de error. Supondremos para simplificar que 0 significa ejecución exitosa. Este mecanismo es equivalente a toda una amplia gama de técnicas de notificación de errores: variables de estado global, parámetros de estado, etcétera, por lo cual nuestra suposición no quitará generalidad a nuestras conclusiones. Partiendo de estas premisas, ¿cómo hay que modificar el código anterior para controlar los posibles errores? La respuesta es la siguiente:

```
if (A() == 0)
    if (B() == 0)
        if (C() == 0)
            if (D() == 0)
                if (E() == 0)
                    return 0;
return -1;
```

Observe que este código, a su vez, debe formar parte de una función creada por nosotros. Y esta rutina hereda el carácter “falible” de las funciones que invoca, por lo que hay que devolver también un valor para indicar si nuestra función terminó bien o

mal. En este caso, hemos simplificado el algoritmo, devolviendo únicamente 0 ó -1 para indicar éxito o fracaso.

Un programador con un mínimo de picardía puede pensar en la siguiente alternativa al código anterior:

```
if (A() == 0 && B() == 0 && C() == 0 && D() == 0 && E() == 0)
    return 0;
else
    return -1;
```

No está mal, se parece un poco al estilo de programación de Prolog²⁶. Pero la función *A* levanta la tapa de un contenedor de sustancias radiactivas y, pase lo que pase, hay que llamar a la función *E*, que cierra el depósito, si *A* terminó satisfactoriamente, ¡y sólo en ese caso, o aténgase a las consecuencias! Está claro que por este camino vamos al holocausto nuclear.

En la práctica, cuando el tema de programación no es tan dramático, lo que sucede es que el código de error no se verifica en todos los momentos necesarios. Es frecuente ver a un programa herido de muerte tambalearse por todo el escenario derribando muebles y tropezando con el resto de los actores. A todo esto nos ha conducido la filosofía “clásica” de verificación de errores.

Contratos incumplidos

La falta de precisión es un pecado, ¿qué queremos decir exactamente con aquello de que “una rutina puede fallar”? Para entendernos vamos a adoptar una metáfora de la programación conocida como *programación por contrato*. Según este punto de vista, una llamada a una rutina representa un contrato que firmamos con la misma. En este contrato hay dos partes: el contratador, que es la rutina donde se produce la llamada, y el contratado, que es la rutina a la cual se llama. El contratador debe proveer ciertas condiciones para la ejecución del contrato; estas condiciones pueden consistir en el estado de ciertas variables globales a la rutina llamada o los parámetros pasados a la misma. Por su parte, el contratado se compromete a cumplir ciertos y determinados objetivos. Es el fallo en el cumplimiento de estos objetivos, o la detección de incumplimientos por parte del contratador, lo que puede causar un error.

El problema de los sistemas tradicionales de control de errores es que el incumplimiento de un contrato por parte de una rutina no utiliza ningún mecanismo especial de señalamiento. Para detectar un contrato incumplido hay que utilizar las mismas instrucciones de control condicionales que cuando se cumplen los contratos, mez-

²⁶ ¿Hay alguien que aún recuerde el *boom* de la Programación Lógica y de los Lenguajes Funcionales?

clándose las situaciones en que van bien las cosas y las situaciones en que van mal. Por este mismo motivo, un contrato incumplido puede pasar desapercibido. Podemos resumir las condiciones mínimas que hay que exigir a una política correcta de control de errores en estos puntos:

- Si quiero llamar sucesivamente a tres funciones, debo poder hacerlo sin mezclar condicionales que contaminen la lógica de mi algoritmo. Debo poder mantener la linealidad del algoritmo original.
- Un error no puede pasar desapercibido.
- Deben respetarse las *condiciones de clausura*: toda puerta abierta debe cerrarse, aún en caso de fallo.
- Los errores deben poder corregirse: un contrato incumplido no debe causar inevitablemente la caída del sistema.

Cómo se indica un error

Entonces vienen las excepciones al rescate. Con nuestro nuevo sistema, cada vez que una función contratada detecta que no puede cumplir con sus obligaciones, produce una excepción. Para producir la excepción se utiliza la instrucción **throw**, que significa “lanzar”. Esta instrucción interrumpe la ejecución del programa y provoca la terminación de todas las funciones pendientes, a menos que se tomen medidas especiales. Al ejecutar la excepción hay que asociarle un valor que porte la información acerca de la situación que la ha provocado. Un ejemplo típico de instrucción para elevar una excepción es el siguiente:

```
throw Exception("¡Algo va mal!");
```

Exception es una clase definida por la VCL, aunque su nombre no comience con la letra *T* (es precisamente la excepción del convenio). En la instrucción anterior se crea “al vuelo” un nuevo objeto de esta clase. El constructor de *Exception* que hemos mostrado tiene un único parámetro, el mensaje explicativo de las causas que motivaron la excepción. El mensaje se almacena en la propiedad *Message* del objeto de excepción construido.

En C++ no es necesario que la información asociada a la excepción se almacene en un objeto. Y si decidimos de todos modos utilizar un objeto, no existen limitaciones sobre el tipo de la clase correspondiente. Estas son instrucciones válidas en C++:

```
throw 2; // No es buen estilo de programación
throw "Toma cadena";
throw TObject();
```

Sin embargo, si utilizamos la VCL debemos ajustarnos al estilo de programación de esta biblioteca, y la VCL utiliza exclusivamente para los objetos de excepciones clases derivadas de *Exception*.

Supongamos que necesitamos una función que divida dos números entre sí. Contratamos la función pasándole un par de enteros, y nos comprometemos a que el divisor que suministramos es distinto de cero. Utilizando el nuevo mecanismo de señalización de fallos, podemos realizar esta implementación:

```
int Dividir(int Dividendo, int Divisor)
{
    if (Divisor == 0)
        // Se ha detectado una violación de las cláusulas del contrato
        throw Exception("División por cero");
    return (Dividendo / Divisor);
}
```

Si en algún momento se llama a esta función con un divisor igual a cero, se ejecutará la instrucción **throw**. Esta instrucción interrumpe el flujo normal de la ejecución del programa: las instrucciones siguientes no se ejecutarán y el programa, si no se toman medidas especiales, debe terminar.

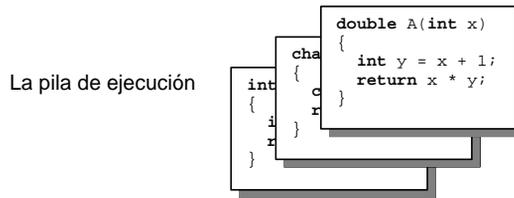
He utilizado como ejemplo la división de dos enteros por simplicidad. En realidad, no hay que verificar explícitamente que el divisor sea distinto de cero, pues C++ Builder captura la interrupción que produce el *hardware* y la transforma automáticamente en una excepción.

La ejecución del programa fluye en dos dimensiones

Voy a recordarle al lector un hecho que, aunque conocido, se olvida con facilidad. Y es que la ejecución de un programa se puede visualizar como un flujo bidimensional. Para poner un ejemplo inmediato, considere cualquiera de los fragmentos de código del epígrafe anterior. Dentro de cada secuencia, descontando los saltos hacia adelante de las condicionales y los saltos hacia atrás de los bucles, las instrucciones siguen una línea de ejecución. ¿Qué sucede cuando se ejecuta la última línea de la secuencia?

¿Caemos al vacío, como imaginaban los medrosos marineros medievales que les sucedía a las embarcaciones que llegaban al fin del mundo?

Pues no: a la función actual la llamé, en su momento, alguna otra rutina, a la cual se regresa entonces. Y a esta rutina, a su vez, la debe haber llamado alguna otra, y así sucesivamente hasta llegar al punto de inicio del programa. La estructura de datos de la aplicación que gestiona esta lista de rutinas pendientes de terminación es la *pila de ejecución* del programa.



Toda cadena es tan fuerte como su eslabón más débil. Si una rutina llama a rutinas propensas a fallar, esta propia rutina es una potencial incumplidora de contratos. Si una de las subrutinas llamadas por esta rutina eleva una excepción, la rutina debe también interrumpir su ejecución.

Una de las ideas principales para comprender las excepciones es que en todo momento el programa está en uno de dos estados posibles: el estado que podemos catalogar de “normal” y el estado de excepción, al cual también podemos llamar “estado de pánico”. En este último estado se produce la interrupción en cadena de las rutinas pendientes en la pila del programa. No existe nada parecido a una excepción sobre otra excepción: o el programa se está ejecutando normalmente o está interrumpiendo rutinas.

Pagamos nuestras deudas

Todo parece indicar que nuestro “revolucionario” sistema para tratamiento de errores se reduce a interrumpir simplemente el funcionamiento del programa. Según esto, el uso de la instrucción **throw** produce casi el mismo efecto que la función *abort* de la biblioteca estándar de C. Si piensa así, se equivoca.

En la descripción de cómo una instrucción **throw** interrumpe la ejecución de un programa hay un detalle importante: la ejecución va interrumpiendo una a una cada función pendiente en la pila de la aplicación. Uno de los objetivos de esta exploración de la pila es la búsqueda de posibles instrucciones de manejo de excepciones: las instrucciones **try/finally** y **try/catch** que estudiaremos en breve. El otro objetivo es la destrucción automática de las variables locales, en especial, de aquellas que pertenecen a una clase.

Tomemos como ejemplo la clase *ifstream*, de la biblioteca estándar de clases de C++, y que se utiliza para leer datos desde un fichero en disco. Supongamos que tenemos la siguiente función:

```

#include <fstream>

int __fastcall SumIntegers(const char* fileName)
{

```

```

    std::ifstream fichero(fileName);
    int sum = 0, i;
    while (fichero >> i)
        sum += i;
    return sum;
}

```

SumIntegers asume que su único parámetro contiene el nombre de un fichero de texto con un valor entero en cada línea; la función suma esos valores y devuelve el gran total. Todo el algoritmo está basado en el uso de la variable *fichero* perteneciente a la clase *ifstream*. Como no se trata de una clase de la VCL, *fichero* se crea directamente en la pila de la aplicación, como una variable automática, y es responsabilidad de C++ liberar la memoria de la variable al finalizar la función. Para esta tarea se llama al destructor definido para la clase que, entre otras cosas, se encarga también de cerrar el fichero físico asociado al objeto.

¿Qué pasa si el algoritmo provoca una excepción en algún punto posterior a la creación del objeto? Según lo que hemos explicado acerca de la propagación de excepciones, se interrumpiría la ejecución normal de la función. ¿Y qué pasa con el objeto creado? Pues que C++ debe garantizar su liberación. El compilador es el encargado de añadir el código necesario para que, en caso de excepción y de forma transparente al programador, siempre se ejecute el destructor de los objetos construidos mediante variables automáticas. Por supuesto, después de ejecutar un destructor implícito por culpa de una excepción, ésta debe seguir propagándose.

En realidad, en el sencillo algoritmo que hemos mostrado es muy poco probable que se produzca una excepción. C++ estándar es un lenguaje “cobarde”, como explicaremos más adelante, y cuando falla un contrato con una función prefiere utilizar alguna bandera o código de error para señalar la situación. Como explicamos al comienzo del capítulo, este comportamiento es funesto para un buen estilo de programación.

La destrucción de objetos dinámicos

El gran problema de la destrucción de objetos temporales consiste en que no todos ellos se crean directamente como variables de pila, sino que muchas veces se utilizan punteros a objetos. Pongamos por caso que la función anterior se hubiera escrito del siguiente modo alternativo:

```

// Mala programación; el uso de punteros es aquí innecesario
int __fastcall SumIntegers(const char* fileName)
{
    std::ifstream *fichero = new std::ifstream(fileName);
    int sum = 0, i;

```

```

while (*fichero >> i)
    sum += i;
delete fichero;
return sum;
}

```

Las reglas de C++ son claras al respecto: los objetos creados en la memoria dinámica deben ser destruidos explícitamente mediante una llamada al operador **delete**. Y si ocurre una excepción dentro de *SumIntegers*, no se libera el objeto, se pierde su memoria y se queda un fichero abierto.

Pero el ejemplo que acabo de mostrar no es muy bueno. A fin de cuentas, ¿para qué diablos necesitamos complicarnos la vida utilizando un puntero a la clase en vez de usar la clase directamente? Ja, ja, ¿ha olvidado entonces que todos los objetos de la VCL deben crearse en la memoria dinámica? Por ejemplo, la VCL permite que configuremos la impresora por medio de la clase *TPrinterSetupDialog*. El siguiente método puede ser una forma de realizar esta tarea, si deseáramos hacerlo mediante un objeto creado dinámicamente:

```

void __fastcall TForm1::ConfigurarImpresora(TObject *Sender)
{
    TPrinterSetupDialog *psd = new TPrinterSetupDialog(NULL);
    psd->Execute();
    delete psd;
}

```

Si se produce alguna excepción durante la ejecución del método *Execute*, perderemos los recursos asignados durante la creación del objeto temporal. ¿Soluciones? Tradicionalmente, C++ ha ofrecido las siguientes:

- Utilizar la instrucción **try/catch**, que estudiaremos en breve. Sin embargo, le adelanto que se trata de una solución chapucera.
- Utilizar punteros “inteligentes”.

¿Cómo es que se mide el coeficiente de inteligencia de un puntero? Con este calificativo, C++ denota clases genéricas que encapsulan un puntero a otra clase. Considere esta sencilla declaración de plantilla:

```

template<class T>
class LocalVCL
{
private:
    T* Instance;
public:
    LocalVCL(T* t) : Instance(t) {}
    LocalVCL() { Instance = new T(NULL); }
    ~LocalVCL() { delete Instance; }
    T* operator->() { return Instance; }
    operator T* () { return Instance; }
};

```

Observe que una instancia de *LocalVCL* almacena un puntero a un tipo genérico *T*, y que al destruirse la clase se destruye también el objeto asociado al puntero interno. Los dos operadores de conversión incluidos nos permiten trabajar con los objetos de la clase *LocalVCL* como si se tratase de punteros al tipo base *T*:

```
void __fastcall TForm1::ConfigurarImpresora(TObject *Sender)
{
    LocalVCL<TPrinterSetupDialog> psd;
    psd->Execute();
}
```

La variable *psd* es ahora en realidad una variable de pila, que internamente contendrá un puntero a un objeto de la clase *TPrinterSetupDialog*. Al terminar la función se destruye *psd*, y por carambola se destruye igualmente el puntero al componente VCL; la destrucción será también automática si ocurre una excepción. Sin embargo, y a pesar de tratarse de una variable de pila, podemos aplicar las funciones miembros de *TPrinterSetupDialog* a la variable *psd*, como podemos comprobar.

Naturalmente, el propio C++ implementa una plantilla de punteros inteligentes, en el fichero de cabecera *memory*, llamada *auto_ptr*. Con la ayuda de esta clase, el método de configuración de la impresora debe escribirse del siguiente modo:

```
void __fastcall TForm1::ConfigurarImpresora(TObject *Sender)
{
    std::auto_ptr<TPrinterSetupDialog> psd(
        new TPrinterSetupDialog(NULL));
    psd->Execute();
}
```

En el resto del libro utilizaremos con bastante frecuencia esta plantilla.

El bloque de protección de recursos

Pero no todo lo que brilla es oro, y cuando se produce una excepción no sólo tenemos que garantizar la destrucción de los objetos temporales. Analice este método:

```
void __fastcall TForm1::OperacionTabla(TObject *Sender)
{
    TablaParadox->LockTable(1tReadLock);
    // Operación larga y peligrosa
    // ...
    TablaParadox->UnlockTable(1tReadLock);
}
```

Debemos suponer que *TablaParadox* es un puntero a un objeto de clase *TTable*. Los métodos *LockTable* y *UnlockTable* imponen y eliminan un bloqueo, en este caso de lectura, sobre la tabla completa. Una vez que se concede el bloqueo, el método realiza alguna operación sobre la tabla que puede potencialmente disparar una excep-

ción. Claro está que en tal caso, a no ser que tomemos medidas especiales, se quedará bloqueada la tabla. Estamos ante otro caso en el que debemos garantizar que, pase lo que pase, se ejecute determinada instrucción. Antes se trataba de una llamada al operador **delete**, pero ahora tenemos un método común y corriente.

C++ estándar no ofrece instrucciones especiales algunas para manejar este tipo de situaciones. Es cierto que con la instrucción **try/catch** podremos poner un mal remiendo al asunto, pero nos costará la duplicación de al menos una instrucción. Pero el estándar de C (sin el “más-más”) sí tiene una instrucción de este tipo, y C++ Builder permite utilizarla en programas escritos en C++. La variante de la instrucción que explicaré será directamente la que se utiliza en el C++ de Borland. Por comodidad, en este libro me referiré a la misma como la instrucción **try/finally**, ignorando el par de caracteres de subrayado que se asocian a **__finally**, y su sintaxis es la siguiente:

```

try
{
    // Instrucciones ...
}
__finally
{
    // Más instrucciones ...
}

```

Si todo va bien y no se producen excepciones, se ejecutan tanto las instrucciones de la parte **try** como las de la parte **__finally**. Esto es: se puede leer la instrucción ignorando las dos palabras reservadas. En cambio, si se genera una excepción durante la ejecución de la parte **try** (que quiere decir “intentar” en inglés) se interrumpe la ejecución del resto de esta parte, pero se pasa el control a la cláusula **__finally**. En tal caso, al terminar las instrucciones **__finally** la excepción sigue propagándose. Por lo tanto, la instrucción **try/finally** no nos permite resolver el problema ocasionado por el incumplimiento de un contrato; nos deja, a lo sumo, pagar a los trabajadores por el trabajo realizado, pero el programa sigue en el modo de pánico.

A la instrucción **try/finally** también se le conoce como el “bloque de protección de recursos”, si por recurso de programación se entiende cualquier entidad cuya petición deba ir acompañada de una devolución. Un fichero abierto es un recurso, pues su apertura debe ir emparejada con su cierre, más tarde o más temprano. Un objeto temporal es otro ejemplo, pues su construcción está balanceada por su destrucción. También son recursos, desde este punto de vista, un bloqueo (*lock*), una tabla abierta, una transacción de bases de datos iniciada, un semáforo para el control de concurrencia... El algoritmo que bloquea una tabla temporalmente debe programarse de la siguiente manera:

```

void __fastcall TForm1::OperacionTabla(TObject *Sender)
{
    TablaParadox->LockTable(ltReadLock);
}

```

```

    try {
        // Operación larga y peligrosa
        // ...
    }
    finally {
        TablaParadox->UnlockTable(ltReadLock);
    }
}

```

La pregunta más frecuente que surge, a la vista del código anterior, es: ¿por qué está la llamada al método *LockTable* fuera de **try**, sin protección? La respuesta es simple: si colocáramos la llamada a esta función dentro de la cláusula **try**, un fallo en la imposición del bloqueo provocaría la ejecución de la cláusula **finally**, y se intentaría devolver un bloqueo que nunca hemos recibido. Este ejemplo de programación se puede generalizar al uso de recursos arbitrarios:

```

// Instrucción que pide un recurso
try {
    // Instrucciones que trabajan con el recurso
}
finally {
    // Instrucción que devuelve el recurso
}

```

Incluso el cambio de cursor durante una operación larga puede considerarse como un caso especial de gestión de recursos. Una vez cambiado el cursor, tenemos que garantizar el retorno a su imagen normal, pase lo que pase en el camino. El siguiente fragmento de código muestra cómo cambiar temporalmente el cursor de la aplicación durante una operación de este tipo:

```

Screen->Cursor = crHourGlass;
try
{
    // Una operación larga que puede fallar
}
finally
{
    Screen->Cursor = crDefault;
}

```

Cómo tranquilizar a un programa asustado

Si solamente tuviéramos las instrucciones **throw** y **try/finally**, y la destrucción automática de objetos de pila, ya habríamos logrado algo importante: la posibilidad de crear programas que fallaran elegantemente. Cuando ocurriese un error en estos programas, el usuario se encontraría de nuevo en el Escritorio de Windows, pero con la seguridad de que todos sus ficheros abiertos se han cerrado y que no se han quedado recursos asignados en el servidor de la red. ¡Menudo consuelo!

Hace falta algo más, hace falta poder corregir el error. Y este es el cometido de la instrucción **try/catch**. En su forma más simple, la sintaxis de esta instrucción es la siguiente:

```
try
{
    // Instrucciones ...
}
catch(...)
{
    // Más instrucciones ...
}
```

Analizaremos la instrucción considerando los dos casos posibles: falla alguna de las instrucciones protegidas dentro de la parte **try** o todo va bien. Si todo va bien, se ejecutan solamente las instrucciones de la parte **try**. En este caso, el programa salta por encima de la cláusula **catch**. Pero si se produce una excepción durante la ejecución del **try**, el programa salta directamente a la parte **catch**, a semejanza de lo que ocurre en **try/finally**. En contraste, una vez finalizada la parte **catch** de la instrucción, la ejecución continúa en la instrucción siguiente a la instrucción **try/catch**: hemos logrado que el programa retorne a su modo de ejecución normal.

Ejemplos de captura de excepciones

Si consideramos las excepciones como modo de señalar el incumplimiento de un contrato, podemos ver a la instrucción **try/catch** como una forma de reintentar el contrato bajo nuevas condiciones. En el siguiente ejemplo intentaremos abrir una tabla; nuestro contrato especifica que, si no existe la tabla, deberemos crearla:

```
try
{
    // Intentar abrir la tabla
    Table1->Open();
}
catch(...)
{
    // Crear la tabla
    CrearTabla();
    // Reintentar la apertura: segunda oportunidad
    Table1->Open();
}
```

Observe que el segundo intento de apertura de la tabla sigue estando fuera de protección. Nuestro contrato no dice nada acerca de pasarnos la vida intentando abrir la maldita tabla. Más adelante explicará por qué no hay que temer a estas situaciones.

Otro tipo frecuente de contrato es aquel en el que debemos reintentar la apertura cíclicamente: la tabla puede estar bloqueada por otra aplicación o puede residir en un

servidor desconectado. En este caso, hay que programar manualmente un ciclo de reintentos; no hay instrucción en C++ que lo haga de forma automática:

```

while (true)
    try
    {
        Table1->Open();
        break;
    }
    catch(...)
    {
        if (MessageDlg("No puedo abrir la tabla\n"
            "¿Reintentar la apertura?", mtError,
            TMsgDlgButtons() << mbYes << mbNo, 0) != mrYes) throw;
    }

```

He utilizado una nueva variante de la instrucción **throw** dentro de la cláusula **catch**. Dentro de las cláusulas **catch** podemos escribir esta instrucción sin especificar el objeto de excepción. En este caso, si el usuario desiste de la apertura del fichero dejamos la excepción sin tratar repitiéndola. ¿Por qué repetimos la excepción, en vez de dar un mensaje aquí mismo y dar por zanjado el tema? Dentro de muy poco daremos respuesta a esto.

Capturando el objeto de excepción

Antes hemos visto que la instrucción **throw** utiliza un valor que sirve para transmitir información acerca de la causa de la excepción. Este valor no siempre corresponde a un objeto, aunque es lo más conveniente; las excepciones de la VCL lanzan siempre objetos derivados de la clase *Exception*. Si necesitamos manejar el objeto de excepción dentro de una cláusula **catch**, para obtener más detalles acerca de la causa de la excepción, la forma de obtenerlo es utilizando la siguiente sintaxis:

```

try {
    // Instrucciones
}
catch(Exception& E) {
    ShowMessage(E.Message);
}

```

La variable local *E* puede ser utilizada entonces en la instrucción asociada a la cláusula **catch**. Observe que el objeto de excepción se debe recibir por referencia, cuando se trata de excepciones generadas por la VCL.

Captura y propagación de excepciones de la VCL

Por razones internas de las técnicas del compilador de C++ Builder, podemos encontrar algunos problemas durante la propagación de excepciones generadas por la VCL. Un poco antes he mostrado un fragmento de código similar al siguiente:

```
try {
    Operacion();
}
catch(...) {
    RepararDestrozos();
    throw;
}
```

Teóricamente, si la primera operación falla con una excepción, se reparan los daños y se relanza la excepción original. Pero en la práctica, el compilador no genera el código correcto y, aunque realmente se lanza una excepción, su tipo original se pierde y se transforma en otra con el mensaje *External exception* o en una violación de acceso, según las condiciones de compilación y de la excepción original.

La solución, no obstante, es sumamente sencilla. Sustituya simplemente la cláusula **catch** para que solamente capture las excepciones de la VCL:

```
try {
    Operacion();
}
catch(Exception&) {
    RepararDestrozos();
    throw;
}
```

Observe que ni siquiera es necesario utilizar una variable para el objeto de excepción. Por supuesto, corremos el riesgo de que los “destrozos” no se reparen si se produce alguna excepción ajena a la VCL, pero eso lo podemos controlar añadiendo finalmente un **catch** sin discriminador de tipo después del que ya existe.

Distinguir el tipo de excepción

La sintaxis completa de la instrucción **try/catch** es un poco complicada, si intentamos distinguir directamente la clase a la que pertenece la excepción. Para esto hay que utilizar varias cláusulas **catch** a continuación de la cláusula **try**. Por ejemplo:

```
try {
    // Instrucciones
}
catch(EDivByZero& E) {
    ShowMessage("División por cero");
}
```

```

    catch(EOutOfMemory&) {
        ShowMessage("Memoria agotada");
    }
    catch(...) {
        ShowMessage("Fallo de excepción general");
    }

```

La primera cláusula **catch** permite trabajar en su interior con el objeto de excepción, nombrado *E*, aunque en este caso no se ha aprovechado el objeto para nada. En la siguiente cláusula, en donde se capturan las excepciones de tipo *EOutOfMemory*, no se declara una variable para recibir el objeto de excepción; si no nos interesa trabajar con dicho objeto, es perfectamente posible. Por último, he ilustrado el uso de una cláusula para capturar los restantes tipos de excepciones, especificando tres puntos suspensivos dentro de los paréntesis. Con esta última cláusula perdemos la posibilidad de trabajar con el objeto de excepciones directamente. La mejor alternativa es utilizar una cláusula **catch** con el tipo general *Exception* al final de la lista de cláusulas de captura, dado que las excepciones más frecuentes en C++ Builder serán las generadas por la VCL:

```

while (true)
    try {
        Table1->Open();
        break;
    }
    catch(EDBEngineError& E) {
        if (MessageDlg("¿Reintentar?", mtError,
            TMsgDlgButtons() << mbYes << mbNo, 0) != mrYes)
            throw;
    }
    catch(Exception& E) {
        ShowMessage(E.Message);
        break;
    }

```

Hay que tener cuidado con el orden de las cláusulas **catch**. Si en el algoritmo anterior invertimos el orden y capturamos primero las excepciones de tipo *Exception* nunca recibiremos excepciones en la segunda cláusula, ¡pues todas las excepciones son de tipo *Exception*!

Una última regla relacionada con **try/catch**: si no escribimos una sección de captura que corresponda a determinado tipo de excepción, esta excepción no se capturará en el caso de que se produzca, y seguirá su destructivo trayecto por la pila de ejecución de la aplicación.

Las tres reglas de Marteens

He dejado a oscuras hasta el momento, intencionalmente, un par de puntos importantes. En primer lugar, ¿por qué, en el algoritmo de “apertura o creación de tablas” dejamos sin proteger la llamada a la rutina de creación? ¿O es que no nos preocupa

que falle la segunda oportunidad? ¿Por qué, en segundo lugar, cuando el usuario decide no reintentar la apertura del fichero volvemos a lanzar la excepción original?

Cuando imparto cursos de C++ a personas que nunca han trabajado con excepciones, me gusta exponer un conjunto de tres reglas simples para la correcta aplicación de esta técnica. Son reglas pensadas fundamentalmente para el programador novato; un programador avanzado descubrirá numerosas excepciones, sobre todo a la tercera regla. Hay una regla por cada tipo de instrucción, y cada regla se refiere al uso y frecuencia que se le debe dar a la misma. Estas son las reglas:

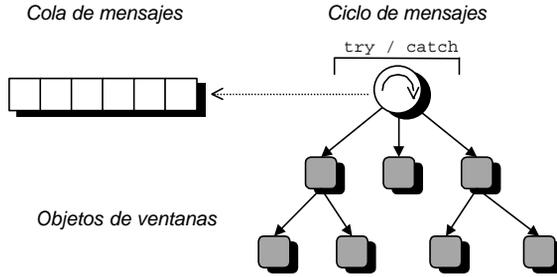
- ❶ (La regla de **throw**) Utilice **throw** con toda naturalidad, cada vez que quiera evitar que el algoritmo siga su ejecución lineal, cada vez que detecte una situación que no puede manejar en ese momento. En pocas palabras, cada vez que se le antoje.
- ❷ (La regla de **try/finally**) Utilice **try/finally** cada vez que pida un recurso, para asegurar su devolución. Esta es la regla de la “honestidad”.
- ❸ (La regla de **try/catch**) Utilice **try/catch** lo menos posible; en la práctica, casi nunca.

La tercera regla, como se puede ver, es paradójica. Es fácil comprender la primera: la invocación a **throw** es el reconocimiento, por parte nuestra, de la imposibilidad de cumplir el objetivo, o contrato, de la rutina. Todo lo más, nos queda la duda de que nuestro programa pueda funcionar con un uso tan “alegre” de esta instrucción. Por otra parte, la segunda regla es la que garantiza la robustez y estabilidad del programa: todos los ficheros abiertos se cierran siempre, toda la memoria pedida para objetos temporales se libera automáticamente. Pero, ¿es que nunca debemos plantearnos, como pretende la tercera regla, reintentar un contrato incumplido? ¿No estaremos programando aplicaciones que se rinden al enfrentarse al primer problema?

Si estuviéramos escribiendo aplicaciones estilo UNIX/MS-DOS con interfaz de terminal, todas estas dudas serían correctas. Pero el hecho es que estamos programando aplicaciones para Sistemas Controlados por Eventos, y las suposiciones sobre el tipo de arquitectura de estos sistemas hacen variar radicalmente las condiciones de tratamiento de excepciones. La respuesta a los interrogantes anteriores es: no necesitamos instrucciones de captura de excepciones a cada paso porque ya tenemos una en el punto más importante del programa, en el ciclo de mensajes.

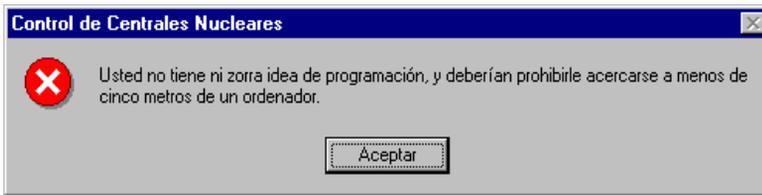
Ciclo de mensajes y manejo de excepciones

El punto más sensible de un programa en Windows es el ciclo de mensajes, y es ahí donde se ha colocado el manejador de excepciones **try/catch** que acabamos de mencionar. La ejecución de una aplicación controlada por eventos se puede resumir en lo siguiente: mientras haya mensajes que procesar, procesémoslos.



Hay un teorema fundamental para estas aplicaciones: la respuesta a un mensaje o evento es virtualmente independiente de la respuesta al próximo mensaje. Por lo tanto, un incumplimiento en un contrato durante el procesamiento de un mensaje solamente debe causar la interrupción de ese proceso, pero el próximo mensaje no debe verse afectado por tal situación.

¿Qué hace la cláusula **catch** del gestor central de excepciones? Muy sencillo: captura el objeto lanzado por la excepción y presenta al usuario el mensaje que contiene en su propiedad *Message*. El método *ShowException* del objeto global *Application* muestra un cuadro de diálogo como el de la figura que muestro a continuación. Un poco más adelante veremos que existe una importante excepción a este comportamiento, y que podemos personalizar la forma en que se presenta el mensaje de la excepción.



La explicación de la tercera regla de Marteens es ahora evidente: si, en un sistema controlado por eventos, se produce un fallo durante el procesamiento de un mensaje y no está en nuestras manos cambiar las condiciones del contrato para reintentar procesar dicho mensaje, lo más sensato es abortar completamente la rama de ejecución actual y pasar al procesamiento del siguiente mensaje. El destino de una excepción que señala el incumplimiento de un contrato debe ser su extinción en la instrucción **try/catch** del ciclo de mensajes. Si antes de llegar a ese punto colocamos una instrucción de este tipo y finalizamos sin cumplir el contrato, estamos defraudando al sistema, lo cual puede ser peligroso.

Para demostrarlo, vamos a suponer que, en el algoritmo de “apertura o creación” colocamos otra instrucción **try/catch** para proteger también el segundo intento de apertura:

```

try {
    Table1->Open();
}
catch(...) {
    try {
        CrearTabla();
        Table1->Open();
    }
    catch(...) {
        // ¿¿¿??
    }
}

```

He puesto a propósito signos de interrogación en el **catch** interno. ¿Qué hacemos en ese punto, llamamos a *ShowMessage*? Bueno, eso ya se hace cuando se captura la excepción en el ciclo de mensajes. Casi siempre algún avispa sugiere entonces transformar nuestra rutina en una función y devolver un error si no podemos abrir ni crear el fichero ... y entonces muestro mi mejor sonrisa irónica: ¿no estamos regresando a la filosofía de tratamientos de errores que tanto criticamos al comienzo del capítulo?

Lo dice la experiencia: cuando el programador novato siente deseos irresistibles de utilizar un **try/catch** es casi seguro que está tratando de convertir un error señalado por una excepción al antiguo sistema de códigos de errores. Por lo tanto, antes de utilizar esta instrucción debe pensarlo dos veces. Este es el sentido de la tercera regla.

Excepciones a la tercera regla de Marteens

Una regla para las excepciones debe tener sus propias excepciones, como es natural. El lector ya conoce una: cuando podemos reintentar el contrato que ha fallado bajo otras condiciones. Otro caso importante sucede cuando queremos realizar alguna labor de limpieza, al estilo de **try/finally**, pero que se ejecute solamente en caso de excepción. Recuerde que la sección **finally** se ejecuta tanto si se producen excepciones como si no las hay. Hay que recurrir entonces a un **try/catch**:

```

try
{
    // Instrucciones
}
catch(Exception&)
{
    LaboresDeLimpieza();
    throw;
}

```

Este uso de **try/catch** no viola el espíritu de las reglas de Marteens: la excepción sigue extinguiéndose en el ciclo de mensajes. La excepción original sencillamente se vuelve a repetir.

Sin embargo, puede haber un motivo más humano y cotidiano para utilizar un **try/catch**: para transformar el mensaje de error asociado a la excepción. Puede ser que estemos trabajando con un C++ Builder en inglés y queramos mostrar los errores en castellano o swahili. Si este es su caso, llame a su distribuidor de software y pregúntele por los precios del Language Pack. Puede también, y esto sí es serio, que el mensaje original de la excepción sea muy general o muy técnico. Para el usuario final, hay que mostrar un mensaje más específico y explicativo. Por ejemplo, cuando en C++ Builder se intenta insertar en una tabla un registro cuya clave está repetida, en pantalla aparece algo tan esclarecedor como “*Key violation*” o “*Violación de clave*”. Muy interesante para el usuario. En estos casos, una solución directa podría ser:

```
try {
    // Instrucciones que pueden fallar
}
catch (const Exception&) {
    throw Exception("Este es mi nuevo mensaje");
}
```

De todos modos la VCL nos ofrece, para la gestión de errores durante el procesamiento de bases de datos, técnicas mejores basadas en la intercepción de eventos, como *OnEditError* y *OnPostError*, que estudiaremos en su momento.

El evento *OnException*

Aunque la captura de la excepción sucede dentro de un procedimiento de la biblioteca VCL, podemos modificar este proceso gracias al evento *OnException* de la clase *TApplication*. El prototipo de este evento es el siguiente:

```
typedef void __fastcall (__closure *TExceptionEvent)
(TObject *Emisor, Exception *E);
```

Si este evento tiene asignado un receptor, el tratamiento convencional de una excepción que llega al ciclo de mensajes no se produce. Este tratamiento consiste simplemente en mostrar el mensaje de la excepción en un cuadro de diálogo, como ya hemos visto. Podemos ser más imaginativos y utilizar un algoritmo alternativo para informar al usuario, quizás utilizando la tarjeta de sonido del sistema (no es broma). No obstante, una aplicación frecuente que se da a este evento es la traducción global de ciertos y determinados mensajes de excepción.

Para imitar el comportamiento normal de C++ Builder, la respuesta a este evento podría ser:

```
void __fastcall TForm1::MostrarExcepcion(TObject *Sender,
Exception *E)
```

```

{
    Application->ShowException(E);
}

```

Una sencilla aplicación de *OnException* es llevar en un fichero de texto el registro de todas las excepciones que llegan a ser visibles para el usuario de una aplicación. Vamos a plantearnos el ejemplo como una unidad que podamos incluir más adelante en cualquier proyecto nuestro. Inicie una aplicación y cree una unidad de código vacía, por medio del comando *File|New* y el icono *Unit*, del cuadro de diálogo que aparece. Guarde el fichero con el nombre que quiera darle a la unidad, y añada los prototipos de los dos métodos siguientes en su cabecera:

```

void __fastcall StartLog(const AnsiString AFileName);
void __fastcall StopLog();

```

Dentro del fichero *cpp* declare e implemente la siguiente clase:

```

class TExceptionLog
{
protected:
    TExceptionEvent SavedHandler;
    AnsiString FileName;
    void __fastcall ExceptionHandler(TObject *Sender, Exception *E);
public:
    __fastcall TExceptionLog(const AnsiString AFileName);
    __fastcall ~TExceptionLog();
};

/* TExceptionLog */
// Aquí vendrá la implementación de la clase TExceptionLog

TExceptionLog *Log = 0;

void __fastcall StartLog(const AnsiString AFileName)
{
    if (!Log)
        Log = new TExceptionLog(AFileName);
}

void __fastcall StopLog()
{
    if (Log)
    {
        delete Log;
        Log = 0;
    }
}

```

Esta unidad exportará los procedimientos *StartLog* y *StopLog*, para activar y desactivar la escritura de las excepciones a disco. *StartLog* puede activarse desde el código inicial de *WinMain*, o en cualquier otro momento, de esta manera:

```
// ...
StartLog(ChangeFileExt(Application->ExeName, ".log"));
// ...
```

Cuando se activa el mecanismo, la unidad crea un objeto de tipo *TExceptionLog* y lo asigna a una variable interna. La idea es que este objeto actúe como receptor del evento *OnException* de la aplicación. Recuerde que un evento es un puntero a un método, y que por lo tanto necesita la presencia de un objeto. No se puede hacer que un evento apunte a una función global.

La implementación del constructor y del destructor de *TExceptionLog* es la siguiente:

```
__fastcall TExceptionLog::TExceptionLog(const AnsiString AFileName)
{
    FileName = AFileName;
    SavedHandler = Application->OnException;
    Application->OnException = ExceptionHandler;
}

__fastcall TExceptionLog::~TExceptionLog()
{
    Application->OnException = SavedHandler;
}
```

Por último, he aquí la implementación del método *ExceptionHandler*.

```
void __fastcall TExceptionLog::ExceptionHandler(TObject *Sender,
Exception *E)
{
    using namespace std;

    Application->ShowException(E);
    try {
        ofstream logFile(FileName, ios_base::app);
        logFile << FormatDateTime("hh:nn:ss", Now()).c_str()
                << " " << E->ClassName().c_str()
                << ": " << E->Message.c_str();
    }
    catch(...) {}
}
```

El método comienza mostrando la excepción al usuario. A continuación, se intentará la grabación de la excepción en el fichero. Pero como el Diablo carga los discos duros, tendremos sumo cuidado de no provocar una segunda excepción que nos lleve de forma recursiva hasta este mismo lugar, para lo cual encerramos todo el bloque de instrucciones dentro de un **try/catch** con la cláusula de recuperación vacía (¡sí, estoy violando la 3ª Regla de Marteens!). La apertura del fichero texto utiliza la opción *app*, para que la escritura tenga lugar siempre al final. Finalmente, escribimos una línea en el fichero que indique la hora actual, el nombre de la clase de excepción y el mensaje asociado. Eso es todo.

La excepción silenciosa

La VCL define un tipo especial de excepción cuya clase es *EAbort*. Desde el punto de vista del lenguaje, este es un tipo de excepción como cualquier otro. Pero la implementación de la Biblioteca de Controles Visuales le otorga un papel especial: si el ciclo de mensajes recibe esta excepción no se muestra mensaje alguno. Es tan frecuente su uso que la unidad *Sysutils* define un procedimiento, *Abort*, que la produce.

Volvamos al caso en que iniciábamos un ciclo de reintentos si fallaba la apertura de una tabla:

```
while (true)
  try
  {
    Table1->Open();
    break;
  }
  catch(EDatabaseError& E)
  {
    if (MessageDlg("¿Reintentar?", mtError,
      TMsgDlgButtons() << mbYes << mbNo, 0) != mrYes) throw;
  }
}
```

La razón por la cual lanzamos un **throw** sin parámetros cuando no queremos reintentar es para no enmascarar la excepción y terminar la rama actual de ejecución dentro del árbol de llamadas. Sin embargo, póngase en el lugar del usuario, que recibe entonces la siguiente secuencia de mensajes:

```
El Ordenador: Ha fallado la apertura de manias.txt. ¿Reintentar?
El Usuario: Bueno, sí.
El Ordenador: Ha fallado nuevamente la apertura de manias.txt.
               ¿Reintentar?
El Usuario: Creo que no.
El Ordenador: Por tu culpa ha fallado la apertura de manias.txt.
```

El último mensaje sobra; lo único que queríamos era *abortar* la secuencia de funciones pendientes en la pila de llamadas, no culpabilizar al usuario de un error que ya conoce. La solución es, claro está, terminar con *Abort*, no con **throw**:

```
while (true)
  try {
    Table1->Open();
    break;
  }
  catch(EDatabaseError& E) {
    if (MessageDlg(E.Message + "\n¿Reintentar?",
      mtError, TMsgDlgButtons() << mbYes << mbNo, 0) != mrYes)
      Abort();
  }
}
```

De este modo, cuando lanzamos una excepción después de mostrar un cuadro de mensajes, es conveniente utilizar *Abort*, para evitar la posible confusión del usuario al recibir múltiples mensajes.

Constructores y excepciones

Una característica fundamental de C++ es la destrucción automática de un objeto cuando se produce una excepción durante su construcción. Supongamos que, en una aplicación MDI, tenemos ventanas hijas que muestran mapas de bits cargados desde ficheros gráficos. Para poder ser precisos, digamos que la clase de ventanas hijas se denomina *THija*, y que cada ventana de este tipo tiene un control de imágenes, *Image1*, en su interior:

```
class THija : public TForm
{
    // ...
    TImage *Image1;
    // ...
};
```

La ventana principal debe crear dinámicamente ventanas de este tipo, en respuesta al comando *Fichero|Abrir*; en realidad debe crearlas y además cargar el fichero dentro del control:

```
void __fastcall TPrincipal::Abrir1Click(TObject *Sender)
{
    if (OpenDialog1->Execute())
    {
        THija *Hija = new THija(Application);
        Hija->Image1->Picture->LoadFromFile(OpenDialog1->FileName);
    }
}
```

¿Qué sucede si no se puede cargar la imagen desde el fichero? Pues que la ventana recién creada quedará abierta, pero su interior quedará vacío. ¿Qué tal si intentamos cargar primero el fichero en una variable temporal, y luego creamos la ventana? Da lo mismo, porque la creación de la ventana también tiene razones para fallar y dejarnos con un objeto creado en memoria. Por supuesto, para solucionar estos problemas tenemos las instrucciones **try**. Pero podemos hacer algo mejor. ¿Por qué no reconocer que la creación de la ventana y la inicialización de su interior es un proceso atómico? Y digo atómico para indicar que se realizan las dos operaciones o no se realiza ninguna. Podemos crear un nuevo constructor en la clase *THija*:

```
__fastcall THija::THija(TComponent *AOwner,
    const AnsiString AFileName) : TForm(AOwner)
{
    Image1->Picture->LoadFromFile(AFileName);
}
```

y llamar a este constructor para crear la ventana:

```
void __fastcall TPrincipal::Abrir1Click(TObject *Sender)
{
    if (OpenDialog1->Execute())
        new THija(Application, OpenDialog1->FileName);
}
```

Aparentemente no hemos ganado nada, excepto desde el punto de vista filosófico. Y es que todavía no he explicado que los constructores definen en su interior una instrucción **try/catch** implícita. El constructor anterior equivale aproximadamente al siguiente código:

```
__fastcall THija::THija(TComponent *AOwner,
    const AnsiString AFileName) : TForm(AOwner)
{
    try {
        Image1->Picture->LoadFromFile(AFileName);
    }
    catch(...) {
        delete this;
        throw;
    }
}
```

Si se produce ahora un error durante la carga del fichero, se ejecuta automáticamente el destructor del objeto: un objeto se construye por completo o no se construye en absoluto. Además, dentro del propio objeto solamente se destruirán los objetos subordinados que hayan podido realmente ser creados. Pongamos por caso que la ventana contiene un objeto de alguna clase que no pertenezca a la VCL, como la *ifstream* que sirve para abrir en modo de lectura un fichero del sistema operativo:

```
class TVentana : public TForm
{
protected:
    ifstream Fichero;
public:
    __fastcall TVentana(TComponent* AOwner, AnsiString fileName);
    // ...
};
```

Como *Fichero* es un atributo incrustado dentro del objeto de ventana, debe inicializarse del siguiente modo durante la construcción:

```
__fastcall TVentana::TVentana(TComponent *AOwner,
    AnsiString fileName) :
    TForm(AOwner),
    Fichero(fileName.c_str())
{
    // ...
}
```

En este caso, si la excepción ocurre durante la ejecución del constructor heredado de *TForm*, el objeto *Fichero* no ha sido construido todavía, por lo que C++ no intenta destruirlo. Pero si el error se detecta dentro de las llaves, entonces sí hace falta destruir a *Fichero*.

¿Y qué pasa en el caso más común en C++ Builder, cuando la clase guarda punteros a los objetos subordinados? Supongamos que el constructor de *THija*, en vez de cargar el fichero en un control *TImage*, crea un mapa de bits en memoria, un objeto de la clase *TBitmap*:

```
class THija : public TForm
{
    // ...
    Graphics::TBitmap *Bitmap;
    // ...
public:
    __fastcall THija(TComponent *AOwner, const AnsiString AFileName);
    __fastcall ~THija();
};

__fastcall THija::THija(TComponent *AOwner,
    const AnsiString AFileName) : TForm(AOwner)
{
    Bitmap = new Graphics::TBitmap;
    Bitmap->LoadFromFile(AFileName);
}

__fastcall THija::~THija()
{
    delete Bitmap;
}
```

Si falla la lectura de fichero se ejecuta el destructor, y se destruye la instancia dinámica apuntada por la variable *Bitmap*. ¿Y si falla la creación del propio objeto *Bitmap*? En ese caso, esta variable tendrá el valor inicial *NULL*, y tenemos que cuidar que no se libere el objeto si encontramos este valor en *Bitmap*. Pero de eso se encarga el operador **delete**, que no invoca al destructor si encuentra que el puntero que se le ha suministrado es un puntero vacío.

CODA

NO ESTARÍA BIEN QUE ME DESPIDIERA del lector sin más; algo de ceremonia me hará parecer más educado. Por ejemplo, podría intentar justificar todo lo que le falta a este libro. Pero después de tantas páginas, me he quedado sin palabras propias. ¿Qué tal si le robo un par de estrofas al viejo Lao Tse?

Treinta radios se juntan en el cubo.

Eso que la rueda no es, es lo útil.

Abuecada, la arcilla es olla.

Eso que no es la olla es lo útil.

Traza puertas y ventanas para hacer una habitación.

Lo que no es habitación, ese es el espacio que queda para ti.

*Así que el provecho de lo que es
se halla en el uso de lo que no es*

Lao Tse, Tao Te King

INDICE ALFABETICO

A

Abort, 428, 781
Abrazo mortal, 108, 138
Acciones referenciales, 61
ACID, 127
Active, 230
ActiveForms, 640
ActiveX Template Library, 525
Actualizaciones en caché, 452, 467
Actualizaciones ortogonales, 401, 597
AddIndex, 337
AddObject, 351
AddRef, 517
AfterCancel, 423, 577
AfterClose, 429
AfterDispatch, 628
AfterInsert, 423
AfterOpen, 429
AfterPost, 577
AfterPrint, 654
AfterScroll, 422
Agregados, 504
Alias, 206
 local, 206, 440
 persistente, 206
 persistentes, 463
AliasName, 441
Alignment, 304
AlignToBand, 655
AllowGrayed, 292
alter table, 62, 65
Aplicaciones multicapas, 203
Append, 389, 393
AppendRecord, 395
ApplyRange, 336
ApplyUpdates, 467, 469, 479
AsFloat, 255
AsInteger, 255
Assign, 391
AsString, 255
AutoCalcFields, 260
AutoDisplay, 293
AutoEdit, 232, 282, 296, 303, 389
AutoSize, 287
AutoStretch, 655

B

Bandas, 648, 652
BatchMove, 413
BeforeAction, 318
BeforeClose, 423
BeforeDelete, 303
BeforeDispatch, 627
BeforeEdit, 423, 425, 577
BeforeInsert, 428
BeforeOpen, 231, 372, 429
BeforePost, 114, 424
BeforePrint, 654
BeforeScroll, 422
BeforeUpdateRecord, 596
BeginUpdate, 462
Bibliotecas de Enlace Dinámico
 UDFs para InterBase, 121
Bibliotecas de tipos, 547, 564
 importación, 547
BLOCK SIZE, 35, 209
BlockReadSize, 246
Bloqueos
 en dos fases, 136
 en transacciones, 133, 468
 oportunistas, 210
 optimistas, 399
 pesimistas, 399
Bookmark, 236
Business rules. *Véase* Reglas de empresa
ButtonStyle, 309

C

CachedUpdates, 364, 468
Campos blob, 54, 297, 319
 LoadFromFile, 297
 LoadFromStream, 298
 SaveToFile, 297
 SaveToStream, 298
Campos calculados, 259, 329
 internos, 502
Campos de búsqueda, 261, 263, 308, 329
Cancel, 330, 392
CancelRange, 334
CancelUpdates, 471

- CanModify, 363, 415
- CGI, 621
- ChangeCount, 591
- check, 58
- CheckBrowseMode, 392, 451
- Ciclo de mensajes, 775
- Ciclo de reintentos infinito, 432
- Close, 230
- CloseIndexFile, 326
- CLSID_ShellLink, 510
- CoCreateGuid, 513
- CoCreateInstance, 520
- CoCreateInstanceEx, 521
- CoGetClassObject, 521
- ColInitializeEx, 570
- Columns, 305, 309
- Commit, 449
- CommitUpdates, 469, 479
- Componentes de impresión, 655
- Conjuntos de atributos, 275, 276
- Conjuntos de datos
 - clientes, 493
 - estados, 247, 389, 421
 - fila activa, 234
 - marcas de posición, 235
 - master/detail, 239
 - tablas, 229
- ConstraintErrorMessage, 280
- Constructores, 782
- Consultas dependientes, 370
- Consultas heterogéneas, 362
- Consultas paramétricas, 368
- Consultas recursivas, 172
 - DB2, 196
- Controladores de automatización, 543, 563
- Controles de datos, 232
- Controles dibujados por el propietario, 288
- ControlStyle, 319
- CopyToClipboard, 293
- CreateComObject, 520
- CreateDataSet, 494
- CreateMutex, 397
- CreateOleObject, 560
- CreateTable, 270
- Cursores, 226
 - deallocate, 156
 - especiales del BDE, 697
 - for update, 184
 - Microsoft SQL Server, 155
 - Oracle, 174
 - propiedades, 700
- Cursores unidireccionales, 200
- CurValue, 597

- CustomConstraint, 279
- CutToClipboard, 293

D

- Database Desktop, 73, 81
- Database Explorer, 275
- DatabaseError, 274, 425
- DatabaseName, 229, 361, 440
- DatasetCount, 443
- Datasets, 443
- DB2
 - Control Center, 190
 - índices, 169
 - tipos de datos, 191
 - triggers, 195
- dBase, 37
- DbiBatchMove, 703
- DbiCreateTable, 690
- DbiDoRestructure, 693
- DBIERR_UNKNOWNSQL, 435
- DbiPackTable, 696
- DbiRegenIndexes, 696
- DCL, 49
- DCOM, 203, 527, 579
- DDL. *Véase* SQL
- Decision Cube, 671
- DEFAULT DRIVER, 220, 230
- DefaultDrawColumnCell, 310
- DefaultExpression, 279, 424
- Delegación, 283
- delete, 94
- Delete, 404, 427, 469
- DeleteIndex, 338
- DeleteSQL, 485
- DeleteTable, 270
- Depósito de Objetos, 585, 599
- Diccionario de Datos, 275
 - dominios, 64
 - tipos de datos de MS SQL, 151
- DisableControls, 237
- DisplayFormat, 256, 275
- DisplayLabel, 253, 284, 304
- DisplayValues, 257
- distinct, 78, 85, 89, 363
- DML. *Véase* SQL
- Dominios, 63, 278
- DRIVER FLAGS, 129
- DriverName, 441
- DropDownAlign, 291
- DropDownRows, 291, 308
- DropDownWidth, 291

DroppedDown, 289

E

EAbort, 399, 781
 EDatabaseError, 274, 433
 EDBEngineError, 433
 Edit, 296, 389, 397
 EditFormat, 258
 EditMask, 258, 285
 Editor de Campos, 249, 260, 390
 Editor de Enlaces, 240
 EditRangeEnd, 337
 EditRangeStart, 337
 EmptyTable, 404
 ENABLE INTEGERS, 168
 ENABLE SCHEMA CACHE, 212, 380, 447
 EnableControls, 237, 244
 Encuentro natural, 79
 Encuentros externos, 91
 ENoResultSet, 415
 EOutOfMemory, 774
 Eventos

- de detección de errores, 422, 431
- de transición de estados, 421

 Excepción silenciosa, 428, 781
 Exception, 763
 Exclusive, 231, 404
 ExecSql, 411, 413
 exists, 88
 explain plan, 174

F

Fábricas de clases, 536
 FetchAll, 469
 FetchOnDemand, 590
 FetchParams, 602
 Ficheros asignados en memoria, 207
 FieldByName, 254, 362, 390
 FieldDefs, 268, 270, 322, 494
 FieldName, 253
 Fields, 254, 390
 Fields editor. *Véase* Editor de Campos
 FieldValues, 255, 391
 FILL FACTOR, 209
 Filter, 341
 Filtered, 342, 352
 FilterOptions, 343
 Filtros, 341

- TClientDataSet, 497, 601

 FindField, 254

FindFirst, 352
 FindKey, 323, 327
 FindLast, 352
 FindNearest, 323, 327
 FindNext, 352
 FindPrior, 352
 Flat, 316
 FlushBuffers, 430
 FocusControl, 284
 ForceNewPage, 663
 foreign key, 60, 117
 FormatDateTime, 257
 FormatFloat, 256
 Formularios activos, 640
 Funciones de conjuntos, 83
 Funciones de respuesta del BDE, 702
 Funciones definidas por el usuario, 76, 121

G

GetIDsOfNames, 543
 GetIndexNames, 321
 GetNextPacket, 590
 Global Unique Identifiers. *Véase* Modelo de Objetos Componentes
 GotoCurrent, 235, 359
 GotoKey, 330
 GotoNearest, 330
 grant, 68
 group by, 82, 84, 363

H

HandleShared, 449
 having, 84, 363
 Hint, 316
 holdlock, 155, 183
 HTML, 618

- campos ocultos, 633
- etiquetas, 619

 HTTP, 617

I

identity, atributo, 151
 Image Editor, 315
 ImportedConstraint, 279
 IndexDefs, 270, 322, 494
 IndexFieldNames, 240, 312, 323, 324, 347, 381, 496
 IndexFiles, 325

- IndexName, 240, 324, 326, 381, 496
- Indices, 321
 - creación, 64
 - en DB2, 194
 - en dBase, 332, 336
 - en Oracle, 168
 - reconstrucción y optimización, 66
 - tablas organizadas por, 170
 - TClientDataSet, 503
- inner join, 93
- insert, 94
- Insert, 389, 393
- InsertRecord, 395
- InsertSQL, 485
- InstallShield
 - claves del registro, 716
 - Extensiones, 721
 - fichero de proyecto, 706
 - grupos y componentes, 709
 - información de la aplicación, 707
 - instalación, 705
 - macros de directorios, 708
 - versión profesional, 720
- Integridad referencial, 28, 60
 - Acciones referenciales, 61
 - información, 698
 - propagación en cascada, 427
 - simulación, 116, 159
- InterBase, 41
 - alertadores de eventos, 119
 - check, 58
 - computed by, 56
 - configuración en el BDE, 214
 - excepciones, 117
 - external file, 55
 - generadores, 113
 - procedimientos almacenados, 103
 - recogida de basura, 143
 - roles, 69
 - set statistics, 66
 - tipos de datos, 53
 - UDF, 121
 - usuarios, 66
- InterBase Server Manager, 66
- InterfaceConnect, 551
- Interfaces
 - IClassFactory, 536
 - IConnectionPoint, 550
 - IConnectionPointContainer, 551
 - ICopyHook, 534
 - IDataBroker, 541, 579
 - IDispatch, 541, 585
 - IProvider, 579, 587

- IUnknown, 516
- Interfaces duales, 545
- InterlockedIncrement, 535
- Invoke, 543
- IProvider, 495, 587
- ISAPI, 621
- IsEmpty, 235
- IsMasked, 285
- IsMultiThread, 625
- IsNull, 256, 261

J

- JavaScript, 639

K

- KeepConnections, 444
- KeyExclusive, 334
- KeyPreview, 289

L

- Ley de Murphy, 761
- like, 75, 368
- Listas de acciones, 592
- LoadFromFile, 606
- LoadMemo, 287
- LoadPicture, 293
- Local, 363
- LOCAL SHARE, 207, 713
- Locate, 347, 350, 382
- LockServer, 537
- LogChanges, 499
- LoginPrompt, 445, 604
- Lookup, 330, 347
- Lookup fields. *Véase* Campos de búsqueda
- LookupCache, 263

M

- Maletín, modelo, 606
- Marcas de posición, 235
- Marshaling, 527, 611
- MasterFields, 240
- MasterSource, 240
- MAX DBPROCESSES, 216
- MAX ROWS, 701
- MaxLength, 286
- MaxValue, 274
- Mensajes

CM_GETDATALINK, 319
 MergeChangeLog, 501, 591
 Microsoft SQL Server, 43, 729
 configuración en el BDE, 216
 dispositivo, 146
 filegroups, 149
 Índices, 153
 integridad referencial, 152
 parámetro TIMEOUT, 138
 SQL Enterprise Manager, 145
 tablas temporales, 152
 Midas, 579
 Balance de carga, 582, 609
 Interfaces duales, 610
 MinValue, 274
 ModalResult, 407, 408
 Modelo de Objetos Componentes, 507
 apartamentos, 568, 640
 Bibliotecas de tipos, 547
 GUID, 513
 in-process servers, 526
 Interface Description Language (IDL),
 511
 Interfaces, 509, 514
 marshaling, 527
 proxies, 527
 Puntos de conexión, 550
 Modified, 392, 469, 477
 ModifySQL, 485
 Módulos de datos
 remotos, 584
 Módulos Web, 623
 Motor de Datos, 199
 Administrador, 204
 MTA, 568

N

NET DIR, 208, 464, 713
 NewValue, 485, 597
 Next, 427
 NSAPI, 621
 Null, 77, 342, 349, 391

O

OBJECT MODE, 218, 265
 ODBC, 202, 208
 OldValue, 485, 488, 597
 OLE. Véase Modelo de Objetos
 Componentes
 OleCheck, 518

OLEEnterprise, 203, 579, 608
 OleSelfRegister, 717
 OnAction, 627
 OnAddReports, 662
 OnCalcFields, 248, 260, 329, 348, 421
 OnCellClick, 312, 318
 OnChange, 273, 285
 OnClose, 308
 OnCloseQuery, 407, 450, 476
 OnCreate, 307
 OnDataChange, 295
 OnDeleteError, 422, 431, 487
 OnDrawColumnCell, 309, 318, 472
 OnDrawItem, 288
 OnEditButtonClick, 309
 OnEditError, 399, 422, 431, 577
 OnException, 778
 OnExit, 285
 OnFilterRecord, 248, 341, 346, 421
 OnGetText, 258
 OnGetUsername, 605
 OnHTMLTag, 630
 OnLogin, 445, 605
 OnMeasureItem, 288
 OnNewRecord, 183, 279, 422, 423
 OnPassword, 465
 OnPopup, 326
 OnPostError, 400, 422, 431, 437, 487
 OnPrint, 660
 OnReconcileError, 599
 OnSetText, 258
 OnStartup, 464
 OnStateChange, 294
 OnTitleClick, 312
 OnUpdateData, 295
 OnUpdateError, 422, 487, 596
 OnUpdateRecord, 422, 484, 486
 OnValidate, 273, 421, 424
 Open, 230
 OpenIndexFile, 326
 Optimización de consultas
 Oracle, 173
 Oracle, 45, 168, 733
 clusters, 170
 configuración en el BDE, 217
 connect by, 173
 cursores, 174
 packages, 179
 Procedimientos almacenados, 171
 secuencias, 182
 SQL*Plus, 163
 tipos de datos, 166
 tipos de objetos, 184

triggers, 176
order by, 81, 234, 363
organization index, 170
outer join. *Véase* Encuentros externos

P

Packages
 Instalación, 713
 Oracle, 179
 QuickReport, 649
PacketRecords, 590
Paradox, 34, 35
 configuración de la caché, 211
 empaquetamiento, 695
 integridad referencial, 695
ParamByName, 369, 418
Params, 369, 418, 441
PASSWORD, 445
PasteFromClipboard, 293
PATH, 442, 712
Perro
 de Codd, 26, 31, 436
PickList, 308
Pila de ejecución, 764
Post, 247, 392, 469
Preparación de consultas, 371
Prepare, 371
primary key, 59
Privilegios, 68
Procedimientos almacenados, 34, 99, 460
ProviderFlags, 595
Punteros inteligentes
 a interfaces, 521
Puntos de conexión, 550

Q

QueryInterface, 517, 518
QuickReport, 557, 647
 Componentes de impresión, 655
 expresiones, 656
 Informes compuestos, 662
 Informes con grupos, 657
 Informes master/detail, 661
 plantillas y asistentes, 649
QuotedStr, 345

R

Rangos, 321, 333
ReadOnly, 231, 282

RecNo, 497
Reconciliación, 599
RecordCount, 235, 427
Refresh, 336, 449
RefreshLookupList, 263
Registro de transacciones
 Paradox, 36
Registros de transacciones, 134
Reglas de empresa, 421
Reglas de Marteens, 431, 775, 777
Rejillas de selección múltiple, 313
Relación de referencia, 261
Relación master/detail, 62, 239, 261, 323,
 358, 428, 452, 471, 602, 661
 Midas, 601
Release, 517
RenameTable, 271
Replicación, 140
ReportSmith, 648
RequestLive, 363, 379
Required, 274
ResetAfterPrint, 657
Resolución, 589, 594
resourcestring, 316
Restricciones de integridad, 26, 57
 clave artificial, 59
 clave externa, 28, 60
 clave primaria, 27, 59
 claves alternativas, 59
revoke, 68
Rollback, 449
RowsAffected, 412
ROWSET SIZE, 218

S

SavePoint, 498
SaveToFile, 606
SCHEMA CACHE DIR, 447
select, 74
 selección única, 87
 subconsultas correlacionadas, 88
SelectedField, 307
SelectedIndex, 307
SelectedRows, 313
SendParams, 602
Serializabilidad, 131
SERVER NAME, 214
Servidores de automatización, 541
Sesiones, 226, 453
Session, 453
Sessions, 453

SetFields, 396
 SetLength, 394
 SetRange, 334
 SetRangeEnd, 336
 SetRangeStart, 336
 ShowException, 776
 ShowModal, 351
 SQL
 Lenguaje de Control de Datos, 49
 Lenguaje de Definición de Datos, 49
 Lenguaje de Manipulación de Datos, 49,
 94
 SQL Links, 52, 201
 SQL Monitor, 377
 SQL Net, 217
 SQLPASSTHRUMODE, 441
 SQLQRYMODE, 362
 STA, 568, 640
 StartTransaction, 449, 451, 477
 State, 247, 389, 424
 Stored procedures. *Véase* Procedimientos
 almacenados
 StoredDefs, 494
 StoredProcName, 418
 suspend, 106

T

Tablas
 anidadas, 225
 mutantes, 177
 Tablas anidadas
 Midas, 601
 TableName, 229
 TableType, 230, 270
 TActionList, 592
 TADTField, 265
 TAggregateField, 252
 TApplication, 778
 TArrayField, 265
 TBDEDataSet, 224
 TBlobStream, 298
 TBookmark, 235
 TBookmarkList, 313
 TBookmarkStr, 236
 TBooleanField, 255, 257, 292
 TCalendar, 293
 TChart, 672
 TClientDataSet, 493, 580, 590
 MergeChangeLog, 591
 SendParams, 602
 TColumn, 305
 TControlClass, 277, 285
 TCriticalSection, 574, 586
 TCustomMaskEdit, 285
 TCustomResolver, 594
 TDataAction, 432
 TDatabase, 206, 229, 277, 439, 449, 469
 herencia visual, 448
 InTransaction, 449
 IsSqlBased, 440
 Temporary, 439
 TransIsolation, 133
 TDataLink, 283
 TDataSet, 223
 TDataSetField, 252, 265, 309, 601
 TDataSetPageProducer, 759
 TDataSetProvider, 594
 TDataSetTableProducer, 631
 TDataSource, 232, 281, 294, 315, 389
 TDBChart, 671, 676
 series, 676
 TDBCheckBox, 284, 292
 TDBComboBox, 287
 TDBCtrlGrid, 318
 TDbDataSet, 418, 471
 TDBDataSet, 225
 TDBEdit, 284, 285
 OnChange, 285
 TDbGrid, 304
 TDBGrid, 233
 ShowPopupEditor, 309
 TDbGridColumn, 305
 TDBImage, 284, 293, 319
 TDBListBox, 287
 TDBLookupComboBox, 263, 284, 290, 308,
 323
 TDBLookupListBox, 290, 323
 TDBMemo, 284, 286, 319
 TDBNavigator, 233, 314
 TDBRadioGroup, 287, 293, 319
 TDBRichEdit, 286, 319
 TDBText, 287
 TDCOMConnection, 589, 606, 609
 TDecisionCube, 677
 TDecisionGraph, 677
 TDecisionGrid, 677, 679
 TDecisionPivot, 677, 679
 TDecisionQuery, 677
 TDecisionSource, 677
 TDrawGrid, 304
 TField, 251
 IsNull, 256
 Name, 253
 OldValue, 488

- OnChange, 273
- Value, 255
- TFieldClass, 277
- TFieldDataLink, 283
- TFieldDef
 - ChildDefs, 270
- TFieldDefs, 268
- TGraphicControl, 287
- throw, 763, 765, 775
- TLabel, 284
- TLineSeries, 673
- TMemoryStream, 298, 669
- TNestedTable, 242, 265
- TNumericField, 251
- TObjectField, 252
- TPageControl, 675
- TPageProducer, 629
- TPopupMenu, 326, 343
- TProvider, 588
- TQRCompositeReport, 662
- TQRDetailLink, 661
- TQrExpr, 656
- TQRGroup, 658
- TQuery, 74, 325, 361, 364, 371, 383, 411, 658, 675
- TQueryTableProducer, 631
- TQuickRep, 658
 - Preview, 650
 - Print, 650
 - PrintBackground, 650
- TQuickReport, 650
- Transacciones, 125, 126, 406
 - aislamiento, 131
 - arquitectura multigeneracional, 139
 - commit work, 128
 - en bases de datos remotas, 602
 - lecturas repetibles, 131
 - rollback work, 128
 - serializabilidad, 131
 - start transaction, 128
- Transaction logs. *Véase* Registros de transacciones
- Transact-SQL, 99
 - triggers, 157
- TransIsolation, 133, 444, 468
- TReconcileErrorForm, 599
- TReferenceField, 265
- TReport, 648
- Triggers, 34, 109, 422, 479
 - anidados, 161, 195
 - new/old, 111, 425, 486
 - Oracle, 176
 - recursivos, 162

- Transact-SQL, 157
- try/catch, 415, 431, 438, 771, 775
 - discriminación de clases, 773
- try/finally, 236, 775
- TSession, 453
 - AddAlias, 463
 - AddStandardAlias, 463
 - ConfigMode, 463
 - DeleteAlias, 463
 - GetAliasDriverName, 460
 - GetAliasName, 460
 - GetAliasParams, 460
 - GetDatabaseNames, 460
 - GetDriverNames, 460
 - GetDriverParams, 460
 - GetStoredProcNames, 460
 - GetTableNames, 460
 - ModifyAlias, 463
 - NetDir, 464
 - PrivateDir, 464
- TSimpleObjectBroker, 609
- TSocketConnection, 608, 609, 642
- TStoredProc, 101, 418
- TStringField, 255
- TStringGrid, 304
- TStrings, 287
- TTabControl, 335
- TTable, 229
 - Constraints, 280
 - FlushBuffers, 430
- TThread, 458
 - FreeOnTerminate, 459
- TUpdateSQL, 364, 485
- TWebActionItem, 626
- TWebDispatcher, 626
- TWebModule, 624

U

- Unassigned, 598
- UndoLastChange, 498
- UniDirectional, 365
- unique, 59
- update, 94
 - problemas en InterBase, 95
- UpdateMode, 400, 484, 488, 595
- UpdateObject, 485
- UpdateRecord, 297
- UpdateRecordTypes, 473
- UpdatesPending, 468, 477
- UpdateStatus, 472, 600
- UpperCase, 346

URL, 618
USER NAME, 445

V

Valores nulos, 77
Valores por omisión
 default, 57, 115
 DefaultExpression, 279
Value, 255
ValueChecked, 292
ValueUnchecked, 292
VarArrayOf, 348
Variant, 255, 348, 391
VarIsNull, 349

VarToStr, 349
Visible, 304
VisibleButtons, 315
Vistas, 66, 96
 actualizables mediante triggers, 181
Visual Query Builder, 373

W

WAIT ON LOCKS, 138
where, 75
WideString, 578
Windows ISQL, 51, 101
with check option, 97